標準MS-DOSハンドブック

MS-DOS HANDBOOK

アスキー出版局

標準MS-DOSハンドブック

アスキー出版局編著

アスキー出版局

商　標
MS-DOSは，米Microsoft社の商標です．
UNIX オペレーティングシステムは AT&T のベル研究所が開発し，AT&T がライセンスしております．
CP／M，CP／M-86は，米Digital Research社の商標です．
Microsoft BASIC，BASCOM，Pascalは，米Microsoft社の商標です．
Lattice Cは，米Lattice社の商標です．

〔Book Cover〕
Design by Kenichi Samura ／ Number One Design Office

はじめに

MS-DOS(MicroSoft Disk Operating System)は，1981年にIBM-PCに採用されたことに端を発し，数多くの16ビットマシンに搭載されてパーソナル・コンピュータ市場に普及してきました．当初リリースされたver. 1.25のMS-DOSは，CP／Mの系譜を引き継ぐ16ビットCPUのDOSであることをその機能に色濃く反映していましたが，現在のver. 2.0では従来の機能に加えてUNIXの影響を強く受けた仕様となり，その機能は大幅に拡張されたものになっています．本書で扱うMS-DOSもこのver. 2.0を中心に解説を行っています．

MS-DOSがなぜ急速に16ビットマシンに普及していったのか，その理由はいくつか挙げることができます．まず，IBM-PCに採用されたこと，他の競合するDOSに比べ価格的に優位に立っていること，マイクロソフト社から優れたアプリケーションが供給されていることなどです．しかし，最終的にいえることは優れたDOSであるからだと思います．日本ではMS-DOSに関する情報が不足しているために，その評価が十分に行われているとはいえませんが，本書が少しでもMS-DOSの理解の助けとなり，数多くのユーザーがMS-DOSを有効に利用する上で役に立つことができればと思っています．

なお，本書はMS-DOSに関する詳細な機能解説を目的としているため，ディスクやCPUなどの基本的な知識はあることを前提としています．これらについては必要に応じて他の文献を参照くださるようお願いいたします．

アスキー・ディスクアルバムのお知らせ

本書に掲載したプログラムを，「アスキー・ディスクアルバム5」として発売しています．

- **5インチ2DD　定価4,300円**
- **5インチ2HD　定価4,800円**
- **8インチ2D　定価4,800円(送料各400円)**

本書の構成

本書は利用目的に応じて3つの部分に分けられており，各部ごとに以下のような章構成になっています．

第Ⅰ部　MS-DOSの機能と操作

第Ⅰ部では，MS-DOSの最も基本的な概念とその構成するソフトウェアの内容と使用法について解説を行います．初めてMS-DOSに接するユーザーあるいは今後MS-DOSを導入しようと考えている方は，まず第Ⅰ部から読みはじめることをお勧めします．

1章　MS-DOSとは

本章では，DOSの一般的な機能およびMS-DOSの特徴についてその概要を述べるとともに，MS-DOSの導入によってコンピュータ・システムにどのようなメリットがもたらされるかについて説明しています．

2章　MS-DOSの操作

MS-DOSがUNIXの影響を強く受けていることから，その仕様は従来のパーソナル・コンピュータのDOSにはなかったいくつかの概念が取り入れられたものになっています．本章ではファイルの操作を中心に，EDLINや各種コマンドの実行例を示すとともに，MS-DOSの特徴的な概念であるデバイスファイル，パイプ，リダイレクト，フィルタ，バッチ処理，環境などについて解説を行います．

3章　MS-DOSとソフトウェア開発

MS-DOSには機械語プログラム開発用のユーティリティ・プログラムが標準で用意されています．本章では実際に機械語プログラムの開発手順を示しながら，LINK，CREF，EXE2BINなどの各種ユーティリティ・プログラムの使用例を示します．

第Ⅱ部　コマンド・リファレンス

第Ⅱ部は，MS-DOSの操作中に引用するためのコマンド・リファレンスとして構成されています．各コマンドは各章ごとにアルファベット順に並べられていますので，後々参照する場合に素早くアクセスすることができます．

1章　MS-DOS コマンド

本章ではMS-DOSのすべての内部コマンドおよび標準的な外部コマンドについて解説を行います．各コマンドは機能の解説とともに実際のサンプル・オペレーションを掲載していますので，コマンドの用途およびその働きを明確に把握することができると思います．

2章　EDLIN

EDLINは，MS-DOSのテキストファイル作成用のラインエディタであり，文書ファイルおよびソースファイルの作成に用いられます．本章はEDLINのコマンド・リファレンスとして利用できるように，すべてのサブコマンドについて，そのオプション指定と実行例を示します．

3章　DEBUG

DEBUGは，MS-DOSの機械語プログラムデバッギングツールであり，機械語プログラミングの開発をサポートするものです．本章では，DEBUGのすべてのサブコマンドについてその使用例と詳細なコメントが付けられており，DEBUGのコマンド・リファレンスとして利用することができます．

第III部　MS-DOSの内部構造

第III部では，MS-DOSの中核となりその制御を行うシステム・ソフトウェアと，MS-DOS上でソフトウェア開発を行うために必要な情報を提供しています．

テクニカルな情報を満載していますので，ソフトウェア開発には欠かせない内容となっています．

1章　MS-DOSのソフトウェア体系

MS-DOSを構成する各種ソフトウェアの概要とその相関関係について説明しています．MS-DOSの中核となる〝MSDOS．SYS〟，〝IO．SYS〟，〝COMMAND．COM〟と外部コマンドの実行がどのような関係を持っているか，また，これらのプログラムがMS-DOSの初期化動作によってどのようにメモリに配置されていくかについて解説を行います．

2章　COMMAND．COM

MS-DOSのコマンド・プロセッサである〝COMMAND．COM〟の働き(リダイレクト，パイプ，内部コマンド)とその処理過程について解説を行います．

3章　MSDOS．SYS

MS-DOSの内部割込み，メモリ管理，プロセス管理，ファイル管理の詳細についての解説を行います．なお，本章でのシステムコールの解説は機能別に分類した概説にとどめてありますので，個々のシステムコールについては7章を参照してください．

4章　IO．SYS

〝IO．SYS〞は，〝MSDOS．SYS〞と外部の周辺装置のインターフェイスを行う部分です．本章では周辺装置の制御を行うデバイス・ドライバを中心に，MS-DOSとデバイス・ドライバの関係，デバイス・ドライバの構造について解説を行います．また本章は，ユーザーが実際にデバイス・ドライバを作成する場合のガイドとなっています．

5章　ファイル・システム

MS-DOSを特徴付ける階層的ディレクトリ構造がどのように実現されているかをディレクトリ，FATの解説を中心に行います．

6章　マクロアセンブラ

MS-DOSの機械語開発ツールであるマクロアセンブラについて，その特徴と使用法について説明しています．マクロアセンブラの仕様は，8ビットCPUのアセンブラに比べると繁雑に思えるほどその機能が豊富であるため，そのすべての機能には触れていません．本章では，通常利用する上で必要な概念，擬似命令等に説明を限定しています．

7章　システムコール実例集

MS-DOSのすべてのシステムコールの解説を行い，各システムコールを用いたサンプル・プログラムに詳細なコメントを付けてあります．

MS-DOS ver. 2.0では従来のCP／Mコンパチブルなシステムコールに加えてXENIXコンパチブルなシステムコールが追加されていますので，これらのシステムコールを利用すれば，MS-DOSの機能を十分に活かし，かつ汎用性の高いソフトウェアを作成することができます．また，各サンプル・プログラムを実行することによって，システムコールの働きをより明確に理解できると思います．

8章　高級言語の紹介

MS-DOSで実行可能な言語コンパイラのうち，BASICとC言語を例にその開発手順を示します．サンプルとして取り上げたプログラムは，階層構造を持ったファイル・システムにおいて有効に利用できるコマンド(wheris, tree)ですので，ソース・プログラムをコンパイルすれば実際に使用することができます．

付録　エスケープシーケンス，漢字コード

本書で利用したシステム構成およびソフトウェア

- PC-9801本体＋8インチ　ディスク・ドライブ2台＋CRTディスプレイ
- PC-9800シリーズ用MS-DOS ver. 2.11
- Lattice Cコンパイラ ver. 2.00
- Microsoft BASIC，Microsoft BASCOM，Microsoft Pascal

目　次

第Ⅰ部　MS-DOSの機能と操作……11

1章　MS-DOSとは……12
1.1　MS-DOSの誕生……12
1.2　DOSはシステム・ソフトウェア……13
1.3　MS-DOSのもたらす環境……15
1.4　MS-DOSの導入と実行環境……17
1.5　MS-DOSの構成……19

2章　MS-DOSの操作……24
2.1　コマンド操作にかかわる基本事項……24
2.2　ファイル操作……30
2.3　ディスクとファイル管理……50
2.4　デバイスファイル……57
2.5　フィルタとパイプ……61
2.6　バッチ処理……67
2.7　コマンドと環境……73

3章　MS-DOSとソフトウェア開発……75
3.1　MS-DOSのプログラム開発用ユーティリティ……75
3.2　MS-DOS上でのプログラム開発……77
3.3　プログラム開発の実例……78

第II部　コマンド・リファレンス……87
コマンド・リファレンスの形式……88
コマンドの書式指定……88
サンプル・オペレーションについての注意……89

1章 MS-DOSのコマンド……90

BREAK……92　CHDIR(CD)……93　CHKDSK……95
CLS……97　COPY……98　CTTY……102
DATE……104　DEL(ERASE)……106　DIR……107
DISKCOPY……110　EXIT……112　FIND……115
FORMAT……117　MKDIR(MD)……119　MORE……121
PATH……123　PRINT……125　PROMPT……127
RECOVER……129　REN(RENAME)……131　RMDIR(RD)……133
SET……134　SORT……137　SYS……139
TIME……140　TYPE……141　VER……143
VERIFY……144　VOL……145

バッチコマンド……146

ECHO……146　FOR……147　GOTO……150
IF……151　PAUSE……153　REM……155
SHIFT……157

2章 EDLIN……158

Append……160　Copy……163　Delete……164
edit……165　End……166　Insert……168
List……168　Move……171　Page……172
Quit……173　Replace……174　Search……176
Transfer……177　Write……178

3章 DEBUG……180

Assemble……183　Compare……184　Dump……185
Enter……186　Fill……187　Go……188
Hex……191　Input……191　Load……192
Move……194　Name……194　Output……196
Quit……197　Register……197　Search……199
Trace……200　Unassemble……202　Write……204

第III部 MS-DOSの内部構造……207

1章 MS-DOSのソフトウェア体系……208
1.1 MS-DOSの基本構成……208
1.2 MS-DOSのメモリマップとブート手順(起動のメカニズム)……210

2章 COMMAND.COM……218
2.1 コマンドの入力と実行……218
2.2 I/Oのリダイレクト……223
2.3 パイプ……225
2.4 COMMAND.COMの再ロード……228

3章 MSDOS.SYS……230
3.1 MSDOS.SYSの概要……230
3.2 内部割込みの機能……231
3.3 MS-DOSにおけるプログラムの実行(メモリモデル, PSP, 環境)……235
3.4 プログラムの実行開始と終了……249
3.5 プログラムのチェインと中断……254
3.6 FCBとファイル・ハンドル……259
3.7 システムコール……271
3.8 システムコールとMSDOS.SYSの内部処理……289

4章 IO.SYS……307
4.1 IO.SYSとデバイス・ドライバ……307
4.2 デバイス・ドライバの呼出し……316
4.3 BPBとメディアチェック……320
4.4 I/Oコマンド……326

5章 ファイル・システム……336
5.1 ディレクトリとファイル……336
5.2 FAT……342
5.3 ディスク上の領域……345

6章　マクロアセンブラ……348
6.1　マクロアセンブラの概要……348
6.2　名前……349
6.3　擬似命令……351
6.4　セグメントの扱い……363
6.5　演算子……368

7章　システムコール実例集……373
7.1　システムコールのコール手順……373
7.2　エラーの処理……374
7.3　レジスタの保存……375
7.4　システムコールの実行例……376
FUNCTION 00H, 01H, 02H……376　FUNCTION 03H, 04H, 05H……378
FUNCTION 06H……381　FUNCTION 07H, 08H, 09H……383
FUNCTION 0AH……388　FUNCTION 0BH, 2AH, 2BH, 2CH, 2DH……390
FUNCTION 0CH……394　FUNCTION 0DH, 0EH, 11H, 12H……395
FUNCTION 0FH, 10H, 14H, 15H, 16H……401　FUNCTION 13H……407
FUNCTION 17H……409　FUNCTION 19H, 30H, 36H, 38H……412
FUNCTION 1AH, 21H, 22H, 23H……417　FUNCTION 24H, 27H, 28H, 29H……422
FUNCTION 25H, 35H……429　FUNCTION 2EH, 54H……431
FUNCTION 33H……434　FUNCTION 39H, 3AH, 3BH……436
FUNCTION 3CH, 3DH, 3EH, 3FH, 40H……440　FUNCTION 41H……447
FUNCTION 2FH, 42H……449　FUNCTION 43H……454
FUNCTION 44H……456　FUNCTION 45H……461
FUNCTION 46H……464　FUNCTION 47H……467
FUNCTION 48H, 49H, 4AH……469　FUNCTION 31H, 4BH, 4CH, 4DH……474
FUNCTION 4EH, 4FH……481　FUNCTION 56H……485
FUNCTION 57H……488

8章　高級言語の紹介……491
8.1　C言語……491
8.2　BASIC……496

付録　エスケープ・シーケンス，漢字コード……506
索引……509

MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK

第I部
MS-DOS の機能と操作

MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK

1章 MS-DOSとは

16ビットマシンがパーソナル・コンピュータの市場に現れるようになって，パーソナル・コンピュータはユーザーの目的に応じてかなり明確に使い分けられるようになってきました．たとえば，8ビットマシンは手軽なゲームマシンあるいはホーム・コンピュータとして新たなユーザー層に受け入れられるとともに，従来の8ビットマシンの能力に不満を持つユーザーは，総合的なシステム環境において優れた16ビットマシンをビジネスユースやソフトウェア開発のツールとして支援するようになっています．

この傾向は，16ビットマシンをサポートしている多くのメーカーが，ディスク・オペレーティング・システム(DOS)を供給するとともに，そのソフトウェアの充実に力を入れていることが原因となっているようです．MS-DOSは，こうした原因を生んだ8086／8088CPU用のシングルユーザー／シングルタスクのディスク・オペレーティング・システムであり，現在では，16ビットマシンの標準的なDOSの地位を占めるに至っています．

本章では，このDOSの導入によってもたらされる一般的なパーソナル・コンピュータの環境について述べるとともに，その環境がMS-DOSではどのように実現されているかについて説明することにします．

1.1 MS-DOSの誕生

1981年に米国でIBM-PCが発表されて以来，パーソナル・コンピュータの市場はしだいに16ビットマシンを主流とするようになってきました．この現象は日本においても例外ではなく，各メーカーとも次々と16ビットマシンを発表してきました．

8ビットマシンから16ビットマシンに移行したことは，単にハードウェアの能力の点で向上しただけではありません．ソフトウェアの分野においても，その種類の広がりと品質の向上には注目するべきものがありますが，その大きな原因の1つとして，DOSの普及が挙げられます．

IBM-PCが発表されたとき，このマシンはPC-DOSと呼ばれるDOSを搭載していました．このPC-DOSが，実は米国マイクロソフト社が開発した8086／8088CPU用のDOS，つまりMS-DOSだったのです．

各メーカーとも，自社の16ビットマシンとIBM-PCとの互換性を保たせるためにMS-DOSを標準採用するようになり，現在多くの16ビットマシンの標準DOSとして世界に広がることになりました．

それでは，なぜDOSの普及がソフトウェアの発展に結びつくのか，その理由について少し考えていくことにします．

1.2 DOSはシステム・ソフトウェア

コンピュータがハードウェアとソフトウェアによって成立している限り，ハードウェアの能力がいかに向上したとしてもその能力を発揮させるソフトウェアがなければ，そのコンピュータ・システムの総合的な能力は低下してしまいます．

ソフトウェアのなかでも，最もハードウェアに密着した部分を受け持つものはシステム・ソフトウェアと呼ばれ，DOSはこのシステム・ソフトウェアを代表するものです．その働きは通常ユーザーがコンピュータを操作する上で意識しないような仕事を受け持っています．たとえば，キーボードから入力した文字をスクリーンに出力したり，ディスクと本体との間でプログラムやデータの入出力を行うなどの処理を行うソフトウェアなのです．

しかし，いま述べたようなことは何もDOSだけに当てはまることではありません．8ビットマシンで主流であったDISK BASICでも，機能的には同じ処理を行います．つまり，DISK BASICもシステム・ソフトウェアと呼ぶことができます．しかしDISK BASICは言語の処理系であるBASICインタープリタに，ディスクを操作する機能を付加したものですから，その能力にはおのずと限界があり，BASIC以外の処理系でのプログラム開発，異機種間でのソフトウェアの互換性，ソフトウェアの充実といった点でユーザーを満足させるものではありません．

1.2.1 ソフトウェアの互換性とその充実

A社のコンピュータで利用できるソフトウェアがB社のコンピュータでは使えないということは，多くの場合そのソフトウェアがA社のコンピュータに依存した部分を持っているためです．たとえば，BASICのプログラムを例にとると，それらはたいてい特定マシンのハードウェア仕様を活かした文法を持っており，それに従ったプログラムであるとすれば異なったハードウェア仕様のコンピュータでは動かないことになります．

異機種間でのソフトウェアの互換性がないことは，ユーザーにとってもまたプログラマーにとっても少しもメリットはありません．ユーザーの立場からすれば，利用可能なソフトウェアの範囲が限定されますし，プログラマーからすれば，各機種ごとに同じ働きをするソフトウェアを開発しなければならないことになり，開発効率が下がってしまいます．また，互換性がなければソフトウェアの蓄積もままなりません．

DOSは，このような異機種間のハードウェア仕様の違いから起こる問題を解消することができます．

つまり，DOSがハードウェアに依存したソフトウェア仕様をすべて吸収することによって，ハ

ードウェアとアプリケーション，ユーティリティといったソフトウェアの緩衝地帯となっているのです．

Fig. 1.1 は，BASIC インタープリタを中心に考えたシステム・ソフトウェアを採用した場合とDOS をシステム・ソフトウェアとした場合の概念的な違いを示しています．

①では，ハードウェアの仕様に BASIC インタープリタが密着しているためにその周辺にはBASIC によるプログラムだけが置かれているのに対し，②，③では，ハードウェアの仕様が異なるにもかかわらず，DOS が緩衝地帯として存在しているためにその周辺のソフトウェアは相互に移動が可能になっています．

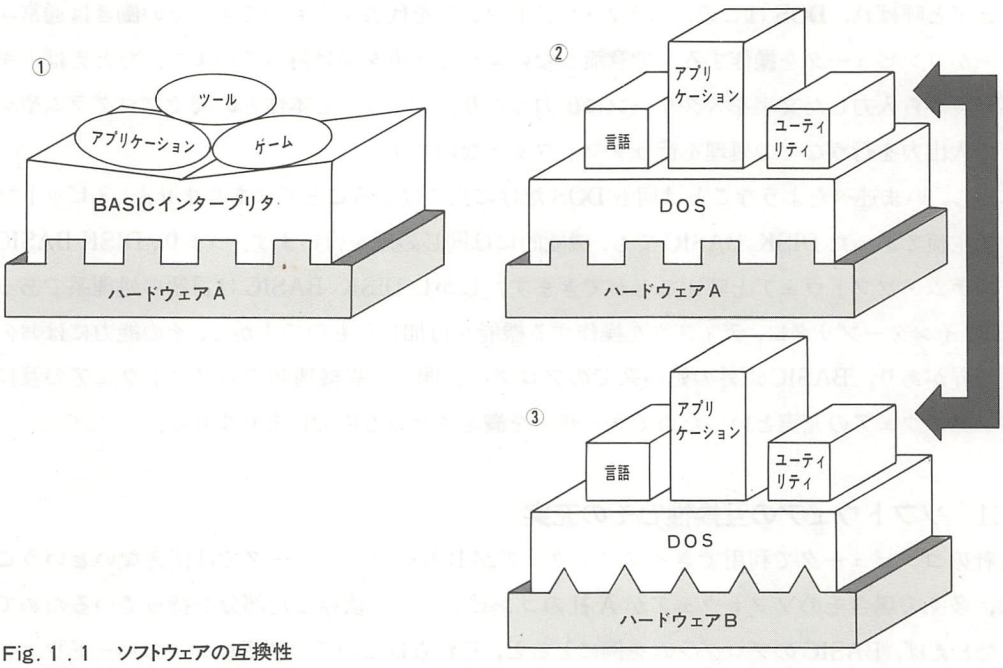


Fig. 1 . 1　ソフトウェアの互換性

このように，DOS は多くのソフトウェアの中間に位置してハードウェアに依存しないソフトウェアの実行環境を用意するのです．

1.2.2　コマンドの実行

DOS はさまざまな働きを行うプログラムによって構成されていますが，それらのプログラムを実行して一定の処理を行うためには，ユーザーは DOS に対してコマンドを実行します．

コマンドの働きには，ファイルの入出力，削除，ファイル名の変更，ファイルのコピー，ファイル名一覧の出力などがあり，コンピュータとユーザー間のマン-マシン・インターフェイスの役割を担っています．

1.2.3　ソフトウェアの開発環境

ユーザーにソフトウェアの開発環境を提供することは，DOSの重要な働きの1つです．

DOSの走る環境では，各種高級言語が利用できるため，さまざまな言語の特徴を活かしたソフトウェアの開発を行うことができます．また，機械語レベルのソフトウェア開発ではアセンブラやデバッガといった開発ツールが備わっているとともに，DOSがユーザーに開放しているサービスルーチン(システムコール)を利用することができるので，ソフトウェア開発を効率よく行うことができます．

1.3 MS-DOSのもたらす環境

MS-DOSは，16ビットマシンのDOSとして前節で述べた機能はすべて備えています．また現在多くの16ビットマシンの標準DOSとして採用されており，ソフトウェアの互換性，充実，開発環境は，16ビットのDOSとしては最も優れた環境を提供しているといえます．そこで，MS-DOSがパーソナル・コンピュータ用のDOSとしてどのような位置にあり，またどのようなソフトウェア環境を提供しているのかについて以下に紹介していきます．

1.3.1　マイクロソフト社のDOS

MS-DOSは，マイクロソフト社の開発したDOSの1つですが，マイクロソフト社はそのほかにもXENIXという16ビットCPU用のOSを供給しています．また，MS-DOSも当初発売されていたver. 1.25からver. 2.0へとバージョンアップしており，その機能は大幅に拡張されています．

Fig.1.2はマイクロソフト社のOSの位置付けを示したものですが，この図で示すようにXENIXは機能的にも価格的にも最も高い位置にあるOSで，パーソナル・コンピュータというよりはむしろミニコンピュータレベルのマルチユーザー／マルチタスクのOSとして製品化されているものです．XENIXの仕様は，UNIX※のver. 7を基本仕様としていますが，さらに機能が拡張されており，サポートされているCPUも8086，Z8000，68000へ移植されています．

これに対しMS-DOSは，シングルユーザー／シングルタスクのパーソナル・コンピュータ用のDOSとして開発されたものであり，とくにver. 1.25のMS-DOSでは8ビットマシンの標準DOSとして市場に広まったCP／M※※の影響を強く受けています．ver. 2.0のMS-DOSは，ver.

※ UNIXは，1970年頃に米国ベル研究所で開発されたミニコン用のマルチユーザー／マルチタスクのOSであり，最初はDECのPDP-7上で作成された．その後PDP-11に移植され，1973年にはアセンブリ言語で書かれていたUNIXの90％がC言語によって書き直されている(ver. 5)．ver. 7は，1978年に発表された拡張バージョン(PDP-11用)である．

※※ CP／Mは，8080，8085，Z80CPU用のシングルユーザー／シングルタスクのDOSであり，8ビット パーソナル・コンピュータの標準的なディスク・オペレーティング・システムとして普及しているものである．

1.25とXENIXの間の機能的なギャップをある程度埋めるもので，主にファイル管理の方法を中心として，XENIXを指向した機能の拡張がなされています．

本書の内容は，主にver. 2.0を対象としていますので，とくに断りのない限り，MS-DOSver. 2.0について述べているものとします．

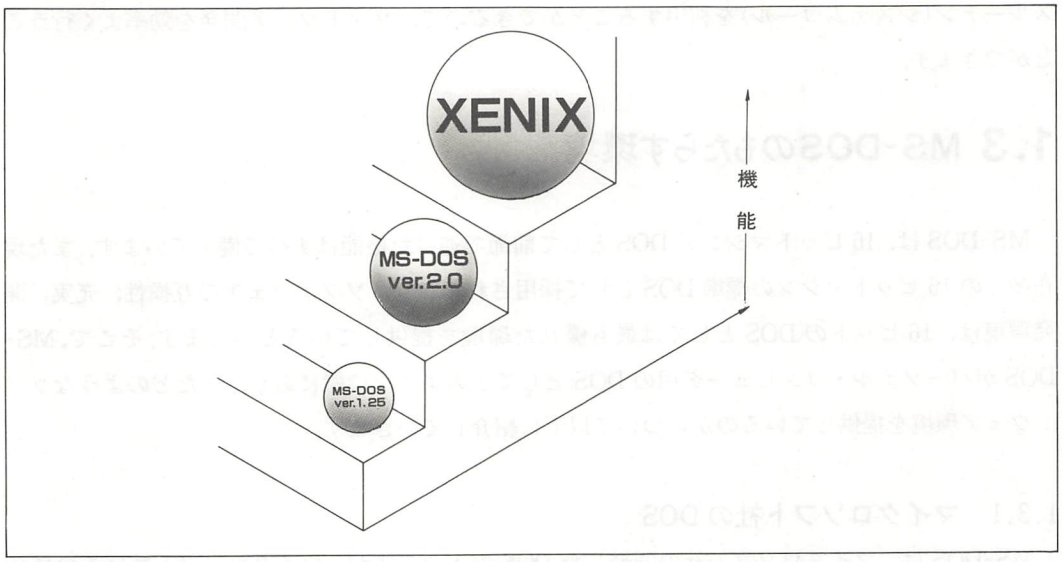


Fig. 1．2　マイクロソフト社のDOS

1.3.2　MS-DOSのソフトウェア環境

MS-DOSが採用されているマシンでは，多種多様なソフトウェアを利用することができます．高級言語としてはBASIC, C, ADA, PROLOG, PASCAL, LISP, FORTRAN, COBOLなど，日頃耳にする言語環境はひととおりそろっているといってよいでしょう．

グラフィックスや漢字処理を伴うアプリケーション・ソフトウェアは，処理速度が要求されることからMS-DOS上で走るといってもハードウェア仕様に依存した部分を持っているものが多く，かならずしも異機種間での互換性が保たれているとはいえませんが，Multiplanというマイクロソフト社が開発した作表言語などは，現在MS-DOSが搭載されているマシンのほとんどにインプリメントされています．

MS-DOSのパッケージには，機械語プログラム開発用のユーティリティ・プログラムとして，アセンブラ，デバッガ，ラインエディタなどが用意されていますが，他のサードパーティからもスクリーンエディタやOSコンバータなどが販売されています．

1.4 MS-DOSの導入と実行環境

前節までの説明で，MS-DOS を導入することによってもたらされる環境については理解していただけたことと思います．

ところで，ユーザーが MS-DOS を導入するにあたり，その購入ルートや必要なハードウェア・システムおよび MS-DOS がサポートする周辺装置などの情報は気になるところです．また，DOS 上のソフトウェアに互換性があるとしても，いったん導入した MS-DOS システム間でのメディアの互換性，つまり MS-DOS が走っている異機種間でディスクが相互に読み書きできるか否かは，ソフトウェアやデータの流通性にとっては重要な問題です．

そこで本節では，MS-DOS を導入する際に必要な情報とともに，マイクロソフト社が標準としてサポートするディスク仕様についての説明を行います．

1.4.1 MS-DOS のリリース

MS-DOS は，マイクロソフト社から各コンピュータメーカー(OEM)に向けて販売されているために，一般のユーザーはマイクロソフト社との個人契約によって購入することはできません．そのため，ユーザーは各機種用にインプリメントされた MS-DOS を各メーカーから購入することになります．

MS-DOS のパッケージの内容は，すべてがマイクロソフト社で作られるわけではなく，その一部は MS-DOS の供給を受けたメーカーで作られます．そのため，MS-DOS の中心となる仕様は同じでも付属のユーティリティ・プログラムやコマンドは少しずつ異なる部分があります(Fig. 1.3)．

```
A>dir /w↵

 ドライブ A: のボリューム・ラベルは 5600JYP
 ディレクトリーは  A:¥
```

IBM5550のシステムディスクの内容

COMMAND	COM	$SYS1Z24	FNT	$SYS1DIC	FNT	$SYSHN24	FNT	$SYSAC24	FNT
$SYS1MAP	FNT	BASIC	COM	FORMAT	COM	CHKDSK	COM	DISKCOPY	COM
PRINT	COM	RECOVER	COM	SYS	COM	MORE	COM	TREE	COM
COMP	COM	DISKCOMP	COM	MODE	COM	EDLIN	COM	DEBUG	COM
USRFNT	EXE	LIB	EXE	FIND	EXE	SORT	EXE	EXE2BIN	EXE
LINK	EXE	DIAG	EXE	ANSI	SYS	CONFIG	SYS		

```
      29 個のファイルがあります
   78848 バイトが使用可能です

A>■
```

```
A>dir /w↵

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥
```

PC-9800シリーズ用のシステムディスクの内容

COMMAND	COM	KNJDIC	SYS	CHKDSK	COM	DEBUG	COM	EDLIN	COM
MORE	COM	RECOVER	COM	SYS	COM	CREF	EXE	EXE2BIN	EXE
FC	EXE	FIND	EXE	LIB	EXE	LINK	EXE	MASM	EXE
SORT	EXE	ASSIGN	COM	COPY2	COM	COPYA	COM	DICM	COM
DISKCOPY	COM	DUMP	COM	FORMAT	COM	KEY	COM	PRINT	COM
SPEED	COM	SWITCH	COM	USKCGM	COM	COPYA	DOC	README	DOC

```
      30 個のファイルがあります
  830464 バイトが使用可能です

A>■
```

Fig. 1.3　PC-9801とIBM5550のシステムディスクの内容

1.4.2　MS-DOS に必要なシステム

MS-DOS を動かすためには，CPU に 8086／8088 が採用された最低 128K バイトのメモリ(RAM)を実装したコンピュータ本体と，ディスク・ドライブ 1 台，CRT ディスプレイが必要です．しかし，このシステムは MS-DOS を利用する上での最低限のシステムであり，プログラム開発などに本格的に利用するためには，実装メモリは 256K バイト以上，ディスク・ドライブ 2 台，そのほかプリンタを備えていた方がよいでしょう．

メーカーが標準システムとして販売しているものは，メモリは 128K バイトしか実装していませんが，高級言語のほとんどは最低 256K バイト実装していないと動きませんし，マクロアセンブラも 256K バイトのメモリがないと動かないシステムがほとんどです．また，アプリケーション・プログラムを利用する上でも，メモリやディスクの容量は大きいほど有利です．本書で利用したコンピュータ・システム(NEC PC-9801)も，256K バイトのメモリ，8 インチディスク・ドライブ 2 台，CRT ディスプレイ，プリンタを標準システムとしてすべてのオペレーションを行っています．

1.4.3　MS-DOS がサポートするシステム

MS-DOS が標準でサポートするデバイス(周辺装置)としては，CRT ディスプレイ，キーボード，フロッピーディスク・ドライブ，補助入出力(RS-232C など)，クロック・デバイスの 5 種類ですが，ハードウェア・システムの仕様によっては RS-232C やクロック・デバイスなどがオプションでサポートされている場合があり，そういったシステムでは，MS-DOS 側のサポートもオプションになっています．

標準以外のデバイスとして，ハードディスクやマウスをサポートしている MS-DOS もありますが，これらは各機種ごとにメーカー側で行っているものです．

また，ユーザー独自のデバイスを MS-DOS に組み込むことができますが，この場合はユーザーがデバイスのためのドライバを書き加えなければなりません．

1.4.4 メディアの互換性

マイクロソフト社では，Table 1.1 に示すような MS-DOS でサポートする標準ディスク・フォーマットを設定しています．これらは互いにフォーマットが異なっていたとしても，同じタイプのディスクであれば，記憶容量が大きい順に上位互換性を持っています．たとえば，倍トラックのドライブでは他のフォーマットのディスクを使用することができます．

Table 1.1 MS-DOSの標準ディスクフォーマット

ディスクの種類	セクタサイズ（バイト）	全容量(バイト)	予約セクタ	セクタクラスタ	ディレクトリ数
8インチ片面単密	128	256K	1	4	68
8インチ両面倍密	1024	1200K	1	1	192
5インチ片面倍密	512	160K	1	1	64
5インチ両面倍密	512	320K	1	2	112
5インチ両面倍密／倍トラック	512	640K	1	2	112

※このTableに示したMS-DOSの標準ディスクフォーマットは一般に普及しているもの(8セクタ／トラック)であり，この他にも5インチディスクには9セクタ／トラックのものもある．詳細は第III部5章参照

1.5 MS-DOSの構成

マイクロソフト社から供給される MS-DOS のパッケージの内容は，Table 1.2 に示すとおりです．これは，さきにも述べたように OEM 契約に基づいて供給されるものであり，各メーカーはこの標準をもとに，自社で販売するコンピュータ・システムに合せてプログラムの追加，削除を行っています．そのため，一般ユーザーが購入する際には Table 1.2 と若干内容の異なる場合(たとえば漢字の辞書ファイルや通信用のプログラムが追加されていたりする)があります．Fig. 1.4 は，PC-9800 シリーズの MS-DOS の内容を示したものですが，黒く塗られた部分が標準的な MS-DOS のプログラムであり，それ以外は NEC がサポートしているものです．

Table 1.2 標準MS-DOSの内容

MSDOS . SYS IO . SYS ※ COMMAND . COM	CREF . EXE DEBUG . COM EDLIN . COM EXE2BIN . EXE FC . EXE LIB . EXE LINK . EXE MASM . EXE※※	FIND . EXE MORE . COM SORT . EXE	CHKDSK . COM DISKCOPY . COM FORMAT . COM SYS . COM RECOVER . COM PRINT . COM

※システムファイルは名称が異なっている機種がある(例：パソピア16，IBM 5550)

※※ MASMは機種によって別売となっている

```
A>dir

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
KNJDIC   SYS     56320  83-09-20  14:22
CHKDSK   COM      6774  83-09-04  23:29
DEBUG    COM     12146  83-09-05   0:35
EDLIN    COM      9124  83-09-17  12:57
MORE     COM      4369  83-09-04  23:44
RECOVER  COM      2607  83-09-04  23:51
SYS      COM       968  83-09-04  23:57
CREF     EXE     13824  82-10-30  16:00
EXE2BIN  EXE      1656  83-09-04  21:14
FC       EXE      2642  83-09-04  23:40
FIND     EXE      6430  83-09-04  23:41
LIB      EXE     32128  82-10-30  16:00
LINK     EXE     42624  83-08-22  15:40
MASM     EXE     80256  83-08-12  14:18
SORT     EXE      1360  83-09-04  23:54
ASSIGN   COM      1069  83-08-12  14:32
COPY2    COM      3312  83-08-27  11:18
COPYA    COM      1319  83-09-13  19:46
DICM     COM      7936  83-09-15  15:30
DISKCOPY COM      6224  83-08-25  13:52
DUMP     COM      1670  83-09-17  19:14
FORMAT   COM     11878  83-10-29  15:46
KEY      COM      4531  83-10-30  19:46
PRINT    COM      4697  83-09-18  18:02
SPEED    COM      1209  83-08-10  15:07
SWITCH   COM      2054  83-10-28  16:30
USKCGM   COM      4130  83-10-30  19:43
COPYA    DOC      1831  83-09-19  15:07
README   DOC      2754  83-10-28  17:22
       30 個のファイルがあります
   830464 バイトが使用可能です

A>■
```

Fig. 1.4　PC-9800シリーズのMS-DOSの内容

それでは，MS-DOSを構成する個々のコマンド(プログラム)を次の3つに大別して説明します．

(1)システム・プログラム
(2)コマンド・プロセッサ
(3)外部コマンド

1.5.1　システム・プログラム(IO．SYS／MSDOS．SYS)

システム・プログラムである〝MSDOS．SYS〟と〝IO．SYS〟の2つのプログラムは，システム・ディスクには書き込まれていますが，ファイル名の一覧を表示してもスクリーンには表示されません．そのため，マニュアルでは〝隠されて見えないファイル〟あるいは〝隠されたファイル〟と表現されています．

これらのプログラムはMS-DOSの中核となるものですが，ユーザーが機械語プログラムの開発あるいはMS-DOSの他機種への移植を考えない限り，意識的に用いることはまずありません．

"MSDOS．SYS"はファイルの入出力，システムコールの処理を中心に行う部分であり，"IO．SYS"は"MSDOS．SYS"の入出力要求によって実際の入出力を行う部分です．後者はメーカーあるいはユーザーによって個々のシステムへのカスタム化が行われます．

1.5.2　コマンド・プロセッサ(COMMAND．COM)

"COMMAND．COM"はユーザーとMS-DOSの本体(MSDOS．SYS)とのインターフェイスを行う部分であり，ユーザーは"COMMAND．COM"と対話しながらDOSに仕事を与えます．換言すれば，"COMMAND．COM"はユーザーの入力したコマンドラインを解析し，対応するコマンドの実行あるいはプログラムの呼出しを行います．

MS-DOSのコマンドのうち，頻繁に利用するコマンドは"COMMAND．COM"の中にサブルーチンとして組み込まれています．これらは内部コマンドと呼ばれ，Table 1.3に示すような種類があります．これらのコマンドはメモリに常駐しているためにコマンドが入力されるとすぐに実行されるものです．たとえばFig. 1.4のPC-9800シリーズのMS-DOSの内容を表示する際に用いた"DIR"は，ディスクのファイル名一覧を出力する内部コマンドです．

Table 1．3　内部コマンド一覧

コマンド	機　能
BREAK	STOP またはCTRL-Cの中断の設定
CHDIR(CD)	ディレクトリの変更およびカレント・ディレクトリの表示
CLS	画面の消去
COPY	ファイルのコピー
CTTY	標準入出力装置(コンソール)を補助装置に変更
DATE	日付の設定／表示
DEL(ERASE)	ファイルの削除
DIR	ディレクトリの表示
ECHO	画面出力の切換え
EXIT	前のプロセスへ戻る
FOR	コマンドの反復実行
GOTO	バッチ処理の制御コマンド
IF	バッチ処理における条件判断
MKDIR(MD)	ディレクトリの作成
PATH	サーチパスの設定および表示
PAUSE	バッチ処理の一時停止
PROMPT	プロンプトの設定
REM	バッチファイルにおける注釈行の指定
REN(RENAME)	ファイル名の変更
RMDIR(RD)	ディレクトリの削除

↓次ページへ続く

コマンド	機　能
SET	環境の設定および表示(環境変数に文字列を代入する)
SHIFT	バッチ処理におけるパラメータのシフト
TIME	時刻の表示／設定
TYPE	ファイルの内容表示
VER	MS-DOSのバージョン番号の表示
VERIFY	ディスクのリード・アフターライトのチェックのセット／リセット
VOL	ボリュームラベルの表示

※(　　)内は別表記

1.5.3　外部コマンド

入力されたコマンドが内部コマンドではなかった場合，"COMMAND．COM"はコマンドに対応するプログラムをディスクからロードして実行します．このようなコマンド(プログラム)を外部コマンドと呼びます．以下の説明では，各コマンドの機能別にその働きを概略的に述べていますが，詳細については第Ⅰ部2章および第Ⅱ部を参照してください．

○ディスクをサポートするコマンド

Table 1.4 に示すコマンドは，MS-DOS で利用するディスクに直接関係するものです．これらのコマンドにより，ディスクの初期化，修復，バックアップなどを行います．

Table 1．4　ディスクをサポートするコマンド

コマンド	機　能
CHKDSK	ディスク，メモリの空き領域の確認およびFATの整合性のチェックと修復
DISKCOPY	ディスク全体のコピー
FORMAT	ディスクの初期化を行う
RECOVER	欠陥セクタを含むファイルから欠陥部分を取り除く
SYS	システムファイル(IO．SYS，MSDOS．SYS)のコピー

○プリンタをサポートするコマンド

プリンタに関する外部コマンドは，次の PRINT コマンドだけです．

Table 1．5　プリンタに関するコマンド

コマンド	機　能
PRINT	ファイルをバックグラウンドでプリントする．補助出力装置への転送も可能

○フィルタ

フィルタとは，入力されたデータを何らかのふるいにかけて出力する働きをするコマンドで，MS-DOS には Table 1.6 に示すように3つのフィルタが用意されています．

Table 1．6　フィルタ・コマンド

コマンド	機　能
FIND	標準入力からデータを読み，指定された文字列を検索する
MORE	入力を1画面ごとに区切って出力する
SORT	ファイルの内容をキャラクタコードの小さい(大きい)順に並べ換える

○ソフトウェア開発コマンド

ソフトウェアの開発に用いられる以下のコマンドは，〝コマンド〟と呼ばれるよりは〝ユーティリティ・プログラム〟と呼ばれるのが普通です．他の外部コマンドと同様にファイル名を入力することによって実行されますが，単にコマンドにパラメータを与えて処理を終了するものだけでなく，コマンド起動後にさらにサブコマンドを用いた処理を行うものもあります．

Table 1．7　ソフトウェア開発コマンド

コマンド	機　能
CREF	アセンブラなどの出力からクロスリファレンスリストを作る
DEBUG	機械語のデバッガ
EDLIN	ラインエディタ
EXE2BIN	EXEファイルをCOMファイルに変換する
FC(FILCOM)	2つのファイルの比較
LIB	ライブラリ・マネージャー
LINK	オブジェクトリンカ
MASM	8086／8088CPUのマクロアセンブラ

2章 MS-DOSの操作

MS-DOSに対する命令は，すべてユーザーが入力するコマンドによって与えられます．ユーザーの立場から考えれば，DOSの持っているコマンドの機能が優れており，単純に処理できるものほど，コンピュータ・システムの操作が容易になります．

本章では，ファイル，ディスク，デバイス，バッチ処理操作にかかわる各コマンドの概念を中心として，オペレーションの実例を挙げながら説明していきます．

なお，各コマンドについての詳細な機能および使用例は，第II部を参照してください．

2.1 コマンド操作にかかわる基本事項

コマンド入力は，キーボードあるいはその他の入力用デバイスによって行われますが，入力の操作が効率よく行えるか否かは，そのDOSのマン-マシン・インターフェイスがどのように考えられているかにかかっています．

たとえば，コマンドラインに長々と入力した文字列のうち，1文字が誤っていたからといって，再度すべての文字を入力しなければならないようでは，マン-マシン・インターフェイスがよくできているとはいえません．また，入力されるDOSのコマンドは，"COMMAND．COM"が解釈，実行できるものでなければなりませんから，入力されるファイル名やドライブ名などは一定の規則に従ったものでなければなりません．本節では，MS-DOSのコマンドレベル(DOSのプロンプトが表示されている状態)の操作を効率よく行うために必要ないくつかの基本事項について説明していきます．

2.1.1 ファイル名の指定

MS-DOSの管理するファイル名は，名前と拡張子によって示されます．ファイル名は，基本的にはユーザーが自由に付けることができますが，Table 2.1に示すように，ファイル名の長さ，予約ファイル名など，いくつかの制約があります．

また一般にファイル名といった場合，名前と拡張子を併せたものを指しますが，名前の部分だけを指してファイル名とすることもあります．

Table 2．1　ファイル名

	名前	拡張子
例	A B C D E F G H	X Y Z
文字数	1～8 文字	3 文字（先頭にかならずピリオドが必要）
使用文字	A～Z　0～9　$　&　#　{ }　~ %　'　( )　－　@　^　`　!　_ カナ　漢字	
予約ファイル名	AUX, CON, PRN, NUL, CLOCK その他各システムで予約されているもの(マニュアル参照)	

実際のファイル名は，Fig. 2.1 の枠で囲まれた部分にあたります．この例は，のちほど解説する DIR コマンドで出力した PC-100 の MS-DOS のシステムディスクの内容です．

```
A>dir /w
 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥

COMMAND  COM   CONFIG   SYS   PC98H    SYS   PC8821   SYS   PC8822   SYS
PC8824   SYS   PC8825   SYS   BOLD     SYS   ITALIC   SYS   AUTOEXEC BAT
VSHELL   COM   VSHELL   LOD   VSHELL   HPT   VSHELL   HPP   MENU     DEF
CHKDSK   COM   DISKCOPY COM   EDLIN    COM   FC       EXE   FIND     EXE
FORMAT   COM   MORE     COM   PRINT    COM   RECOVER  COM   SORT     EXE
LINK     EXE   SYS      COM   HDFORM   COM   LOGOFF   COM   SETUP    COM
SOFTKEY  DAT
        31 個のファイルがあります
     90112 バイトが使用可能です
```

Fig. 2．1　PC-100のMS-DOSシステムディスク

ここで，実際のファイル名に付けられている拡張子は，〝．COM〞と〝．EXE〞〝．SYS〞などになっていますが，これらの拡張子は，MS-DOS において次に述べるような特別な意味を持っています．

拡張子に〝．COM〞(COM ファイル)と〝．EXE〞(EXE ファイル)の付いたファイルは，どちらも MS-DOS における実行型のファイルであることを示しています．つまり，ユーザーがファイル名の名前の部分を入力すれば，コマンドとして実行されるプログラムであるということです．

拡張子に〝．SYS〞の付いたファイル(SYS ファイル)は，MS-DOS がシステム側で利用するファイルを意味していますが，〝．COM〞や〝．EXE〞ほど厳格な意味を持っているわけではありません．ユーザーが作成したファイルに〝．SYS〞の拡張子を付けてもかまいませんが，ファイルの管理という点から考えれば避けた方がよいでしょう．

ここで説明した以外にも，MS-DOS で扱うファイル名の拡張子には特別な意味を持つものがありますが，それらのうちよく使われるものを Table 2.2 に示します．

Table 2.2　拡張子とその意味

拡張子	意　味
ASM	アセンブラのソースファイル
BAK	エディタのバックアップファイル
BAT	バッチファイル
COM	8080メモリイメージの実行型ファイル
EXE	リロケータブルな実行型ファイル
LIB	ライブラリファイル
LST	アセンブラやコンパイラのリスティングファイル
OBJ	アセンブラやコンパイラのオブジェクトファイル
REF	アセンブラのクロスリファレンスファイル
SYS	システムファイル
$$$	テンポラリファイル

2.1.2　ワイルドカード

コマンド操作を行っていると，一度に複数のファイル，あるいはすべてのファイルを対象とした処理を行いたい場合があります．たとえば，ディスク上のファイルの中から拡張子〝．COM〟の付いたファイルのみを選択するような場合が考えられますが，このような場合には，〝？〟や〝＊〟で示されるワイルドカードと呼ばれるキャラクタを利用することによって，次に示すようにファイル名の指定に柔軟性を持たせて処理することができます．

〝？〟は，任意の１文字にあてはまるワイルドカードで，たとえば次に示すようなファイルマッチを行います．

S?M?LE　…SAMPLE, SIMPLE　(2，4文字目がどのような文字でもマッチする)
A??D?　…ABCDE, AABDD　(2, 3, 5文字目がどのような文字でもマッチする)

〝＊〟は，任意の複数文字とマッチします．

TEST.＊　…TEST.COM, TEST.BAK, TEST.$$$などのすべてのファイル名
＊.＊　…すべてのファイル
＊.COM　…すべてのCOMファイル

2.1.3　テンプレート機能によるコマンドラインの修正

キーボードから入力したコマンドやテキストなどを修正する場合，リターンキーを押す前であればバックスペースやカーソルキーを用いて修正を行いますが，ミスタイプに気付かずにリターンキーを押した場合には，テンプレート機能と呼ばれるMS-DOSの編集機能を利用することによって素早く修正することができます．

MS-DOS ではキーボードから入力されたデータは，行単位でいったんテンプレートと呼ばれる入力バッファに格納されます．このテンプレートの内容は，再びコマンドラインへ呼び戻すことができるので，ユーザーが最後に入力したコマンドラインの修正が簡単に行えるようになっています．

テンプレート機能は，Table 2.3 に示す機能を持っており，それらは通常ファンクションキーに割り当てられていますが，機種によってその割当てが異なるため，本書では機能別の省略記号を用いています．

Table 2．3　テンプレート機能

テンプレート機能	機　能
C1	新しい行にテンプレートから１文字コピーする
CU	テンプレートから新しい行へ，指定された文字の直前までをコピーする
CA	新しい行に，テンプレートに残っているすべての文字をコピーする
S1	テンプレート内の１文字をスキップする
SU	指定された文字までテンプレート内の文字をスキップする
VOID	テンプレートの内容を変化させずに現在の入力を取り消す
NWL	現在行の内容をテンプレートに移す
INS	挿入モードにはいる．もう一度押すと挿入モードを解除する

Fig. 2.2 は，テンプレートに〝ABCDEF．XYZ〟という文字列を格納した後に，各テンプレート機能を実行したものです．

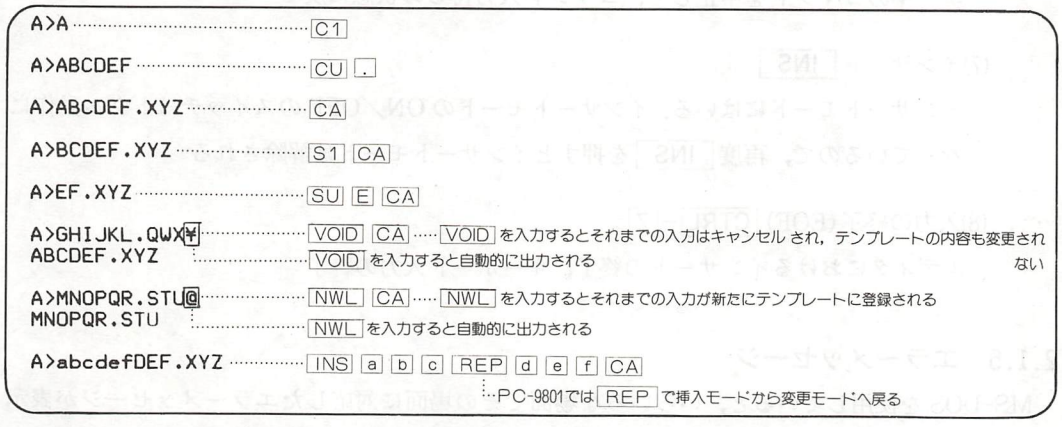


Fig. 2．2　テンプレートの実行例

2.1.4　コントロール・キャラクタの編集機能

コマンドラインの編集機能およびスクリーン表示の中断，スクリーン／プリンタへの同時出力などは，コントロール・キャラクタによって制御することができます．

(1) 1文字削除 CTRL－H，DEL，BS，←
コマンドラインから最後に入力した文字を1文字ずつ削除する

(2) 1ラインキャンセル CTRL－X，VOID
入力した1行をキャンセルし，行末に"¥"を表示して改行(CR，LF)を行う．このとき，テンプレートには影響を与えない

(3)スクリーン・コントロール CTRL－S
スクリーン表示を中断する．任意のキーを押せば再開する

(4)ライン・ターミネート CTRL－J
ラインフィード(LF)

(5)プリンタエコー CTRL－P，CTRL－N
スクリーンに表示されるものを，同時にプリンタに出力する．プリンタエコーのON／OFFスイッチはトグル動作になっているので，再度CTRL-Pあるいは CTRL-N を押すとプリンタエコーは解除される

(6)実行の中断 CTRL－C
実行中のコマンドを中止して，コマンド入力待ちの状態に戻る

(7)インサート INS
インサートモードにはいる．インサートモードのON／OFFのスイッチはトグル動作になっているので，再度 INS を押すとインサートモードは解除される

(8)入力の終了(EOF) CTRL－Z
エディタにおけるインサートの終了，キーボード入力の終了

2.1.5　エラーメッセージ

MS-DOSを使用していると，いろいろな場面でその場面に対応したエラーメッセージが表示されます．エラーメッセージのほとんどはユーザー側の問題である場合が多く，システム側のエラーはほとんど発生しません．もし頻繁にそのようなエラーが発生するようであれば，システム全体のチェックを行う必要があります．

MS-DOS のエラーメッセージには，システム全体に共通しているものと，各コマンドやプログラムに固有のものがありますが，多くの場合エラーメッセージは和文で表示され，またメッセージの内容が具体的な処理方法を明示していますから，メッセージに従って処理を行えば解決できるものがほとんどです．

主なエラーの原因を次に示しておきますが，エラーに関するメッセージの表示は各機種ごとに異なるため，詳細は各機種別のマニュアルを参照してください．

- ファイル名の指定，ドライブ名の指定が間違っている
- ディスクの空き領域が不足している
- メモリの空き領域が不足している
- ディスクが書込み禁止になっている
- コマンドのパラメータが間違っている
- DOS のバージョンが異なっている
- ディレクトリに収容可能なファイル数を超えている
- ディスクの不良

エラーメッセージが出力された後は，通常コマンド待ちの状態になりますので，エラーの種類に応じた処理を行ってください．

2.1.6　バックアップ

MS-DOS に限らず，ディスクを利用する場合に忘れてならないのは，バックアップをとることです．MS-DOS には，RECOVER，CHKDSK コマンドのように，ある程度までディスクの修復をしてくれるコマンドがありますが，これらの処理は完全にディスクを修復するものではありません．重要なデータ，プログラムを収めたディスクは，かならずバックアップをとるようにしてください．

バックアップのとり方としては，〝FORMAT／S〞あるいは SYS コマンドによってシステムファイルをコピーした後に，COPY コマンドで必要なファイルのみをコピーする方法と，DISKCOPY コマンドですべてのファイルをコピーする方法があります．

なお，バックアップをとる手順の詳細については，第II部の各コマンド欄を参照してください．

2.2 ファイル操作

DOSの操作はすべてファイル操作であるといっても，あながちそう大きな誤りではないでしょう，とくにDOSのコマンドレベルの操作では，ファイルにかかわる操作を知らなければ何もすることができません．

そこで本節では，MS-DOSにおけるファイルの概念，ファイル名一覧の出力，ファイルの作成，削除，ファイル名の変更，コピーといった処理について解説していきます．

2.2.1 ファイルの概念

通常ファイルというとディスク上のファイルを指すことが多いようですが，MS-DOSでは，入出力の対象となるものはすべてファイルとして扱われます．具体的にいえば，ディスク上に作られるデータの集まりはもちろん，後述する入出力デバイス——入力デバイスであるキーボード，出力デバイスであるスクリーン，プリンタなど——もデバイス型のファイルとして扱われるのです．

またこれらのファイルにはすべてファイル名が付けられ，MS-DOSはそのファイル名に従ってプログラムを実行したり，データの入出力を行ったりしています．Table 2.1で，デバイス名が予約ファイル名となっていたのも，ディスク上のファイルとの混用を避けるためです．

このように，ファイルの概念を拡張させることによって，MS-DOSではデータの入出力の制御に柔軟性を持たせています(Fig. 2.3)．

なお，本節で扱うファイルはすべてディスク上のファイルに関するものですが，デバイス型のファイルについては2.4において解説します．

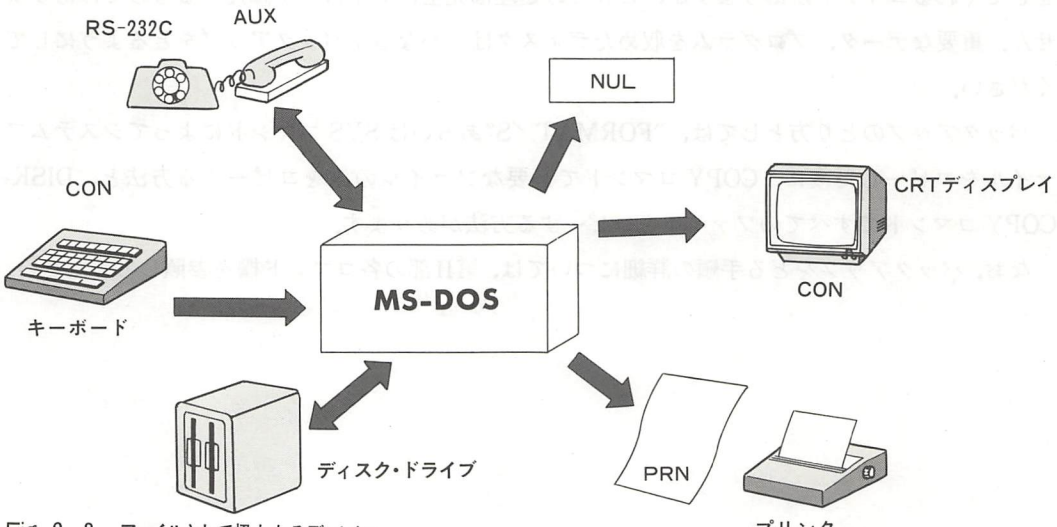


Fig. 2．3　ファイルとして扱われるデバイス

2.2.2　ファイル名一覧の出力(DIR)

ディスクの内容を知るために用いられるDIRコマンドは，DOSのコマンドの中でも最も頻繁に用いられるものです．DIRコマンドを実行すると，ディスクに記録されているファイル名の一覧が出力されます．

Fig. 2.4に示した実行例は，ドライブA(カレント・ドライブ※)にはいっているMS-DOSのシステムディスクの内容を表示したものですが，各ファイルは，ファイル名とともにファイルの大きさ(単位はバイト)，その作成あるいは更新日付／時刻が表示されています．

```
A>dir
  └─カレント・ドライブはドライブA
 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥                          ←ファイルの大きさ(バイト)
                                             ←ファイルの更新日付／時刻
COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
CHKDSK   COM     6774  83-09-04  23:29
DEBUG    COM    12146  83-09-05   0:35
EDLIN    COM     9124  83-09-17  12:57
MORE     COM     4369  83-09-04  23:44
RECOVER  COM     2607  83-09-04  23:51
SYS      COM      968  83-09-04  23:57
CREF     EXE    13824  82-10-30  16:00
EXE2BIN  EXE     1656  83-09-04  21:14
FC       EXE     2642  83-09-04  23:40
FIND     EXE     6430  83-09-04  23:41
LIB      EXE    32128  82-10-30  16:00
LINK     EXE    42624  83-08-22  15:40
MASM     EXE    80256  83-08-12  14:18
SORT     EXE     1360  83-09-04  23:54
ASSIGN   COM     1069  83-08-12  14:32
COPY2    COM     3312  83-08-27  11:18
COPYA    COM     1319  83-09-13  19:46
DICM     COM     7936  83-09-15  15:30
DISKCOPY COM     6224  83-08-25  13:52
DUMP     COM     1670  83-09-17  19:14
FORMAT   COM    11878  83-10-29  15:46
KEY      COM     4531  83-10-30  19:46
PRINT    COM     4697  83-09-18  18:02
SPEED    COM     1209  83-08-10  15:07
SWITCH   COM     2054  83-10-28  16:30
USKCGM   COM     4130  83-10-30  19:43
COPYA    DOC     1831  83-09-19  15:07
README   DOC     2754  83-10-28  17:22
       30 個のファイルがあります
   830464 バイトが使用可能です

A>■
```

PC-9800シリーズ用MS-DOSのファイル名一覧表示
購入時のシステムディスクはこのように表示される

Fig. 2.4　DIRコマンドの実行例

※カレント・ドライブ……プロンプトによって示されているディスク・ドライブ

○ほかのドライブのファイル名一覧表示

MS-DOSでは，複数のディスクを管理することができますが(論理的には63台まで接続可能)，カレント・ドライブを示していても，コマンドの後に特定のドライブ名を指定することによって，任意のドライブのディスクの内容を表示することができます．Fig. 2.5では，カレント・ドライブがドライブAになっている状態で，ドライブBのディスクの内容を表示しています．

```
A>dir b:↵
          ドライブBにはいっているディスクのファイル名一覧を表示する．ドライブ名の後の ":" はかならず付ける
 ドライブ B: のボリュームラベルは ALL_MIGHTY
 ディレクトリは B:¥

COMMAND  COM     17190  83-09-19  16:19
TANJI    SYS     56320  83-09-20  14:22
KNJDIC   SYS    333824  84-04-02  17:02
BIN          <DIR>      84-04-03  19:27
MAN          <DIR>      84-04-03  21:21
BAT      BAK <DIR>      84-04-03  19:32
MP           <DIR>      84-04-03  21:22
AUTOEXEC BAT        36  84-04-06  16:20
CONFIG   SYS        62  84-04-06  16:23
RAMDISK  SYS       505  84-04-11   6:49
      10 個のファイルがあります
  164864 バイトが使用可能です

A>■
```


Fig. 2.5　ドライブBのファイル名一覧表示

○ワイルドカードと／W スイッチの利用

DIRコマンドは，無造作にすべてのファイル名を出力する場合に用いることはあまりないでしょう．むしろ，ある条件にマッチしたファイルがそのディスクにあるか否かを知るために用いる場合が多いので，通常はワイルドカードや／Wスイッチ(ワイド・ディスプレイ)を併用しています．

Fig. 2.6は，拡張子が "*.COM" のファイルをワイド・ディスプレイで表示したものです．

```
A>dir *.com /w↵
          下に表示されているように，ファイル名にすべて "COM" の付いたファイルだけがマッチして表示される
                ワイド・ディスプレイの指定
 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

COMMAND  COM    CHKDSK   COM    DEBUG    COM    EDLIN    COM    MORE     COM
RECOVER  COM    SYS      COM    ASSIGN   COM    COPY2    COM    COPYA    COM
DICM     COM    DISKCOPY COM    DUMP     COM    FORMAT   COM    KEY      COM
PRINT    COM    SPEED    COM    SWITCH   COM    USKCGM   COM
      19 個のファイルがあります
  830464 バイトが使用可能です

A>■
```

Fig. 2.6　ワイルドカードとワイド・ディスプレイの指定

2.2.3　EDLIN によるファイルの作成

DOS の操作のほとんどがファイルにかかわるものであることは前にも述べましたが，ソース・プログラムや文書ファイルの作成あるいは編集を行うためのツールがエディタです．MS-DOS のエディタは，EDLIN と呼ばれるラインエディタで，行単位(1 行 253 文字)でテキストの編集を行うものです．EDLIN は，スクリーン編集機能こそありませんが，2.1 で紹介したテンプレート機能が利用できるため，ほとんどの行内編集はテンプレート機能によってスムースに処理できます．Table 2.4 は EDLIN で利用できるサブ・コマンドの一覧です．

Table 2．4　EDLINのサブコマンド一覧

コマンド	機　能	書　式
A	ファイルからテキストを読み込んで，メモリ上のテキストに追加する	[〈行数〉] A
D	行番号で指定した行，あるいは指定された行範囲を削除する	[〈行番号 1〉][,〈行番号 2〉] D
E	編集作業を終了し，ファイルをディスクにセーブする	E
I	テキストのインサートモードにはいる，または〈行番号〉の前にテキストを挿入する	[〈行番号〉] I
L	〈行番号 1〉から〈行番号 2〉の範囲を表示する	[〈行番号 1〉][,〈行番号 2〉] L
Q	編集を中断する	Q
R	行番号で指定した範囲の〈文字列 1〉を〈文字列 2〉に置き換える	[〈行番号 1〉][,〈行番号 2〉][？] R〈文字列 1〉〈CTRL-Z〉〈文字列 2〉
S	行番号で指定した範囲内で〈文字列〉を検索する	[〈行番号 1〉][,〈行番号 2〉][？] S[〈文字列〉]
W	出力ファイルに〈行数〉で指定したテキストを書き込む	[〈行数〉] W
数字	〈行番号〉を指定し，編集モードにはいる	〈行番号〉
C	〈行番号 1〉,〈行番号 2〉で指定した範囲の行を〈行番号 3〉の直前へコピーする．オプションで〈回数〉を渡す	[〈行番号 1〉],[〈行番号 2〉],〈行番号 3〉,[〈回数〉] C
M	〈行番号 1〉,〈行番号 2〉で指定した範囲の行を〈行番号 3〉の直前に移動する	[〈行番号 1〉][,〈行番号 2〉],〈行番号 3〉M
P	現在行あるいは指定した行から23行を表示する	[〈行番号 1〉][,〈行番号 2〉] P
T	行番号で指定したメモリ上のテキストの直前にファイルを読み込む	[〈行番号〉] T〈ファイル名〉

○新しいテキストファイルの作成

テキストファイルの例として，EDLIN のコマンドリストを入力し，ディスク上にファイルを作成します．ファイル名は，〝EDHLP．TXT〟として EDLIN を起動します．

EDLIN の起動

EDLIN が起動した時点で Fig. 2.7 のように表示されます．〝＊〟は EDLIN のプロンプトです．

新規のファイルを作成する場合には，まだディスクに登録されていないファイル名を指定しなければなりません．その際 EDLIN は，ディレクトリを検索して同じファイル名がなければ新規の

ファイルとして登録しますが，指定したファイル名がすでに登録されているものであった場合には，そのファイルをメモリに読み込みます．

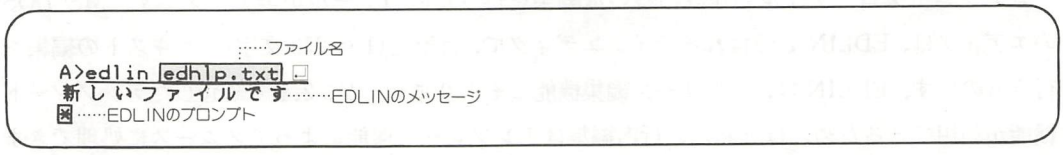


Fig. 2．7　EDLINの起動

テキストの作成(Insert)

まずIコマンドを入力して，テキストのインサートモードにはいります．1行の入力が終了するごとにリターンキーを押し，テキストの入力が終了したら，CTRL-Z を入力してインサートモードを抜けます．

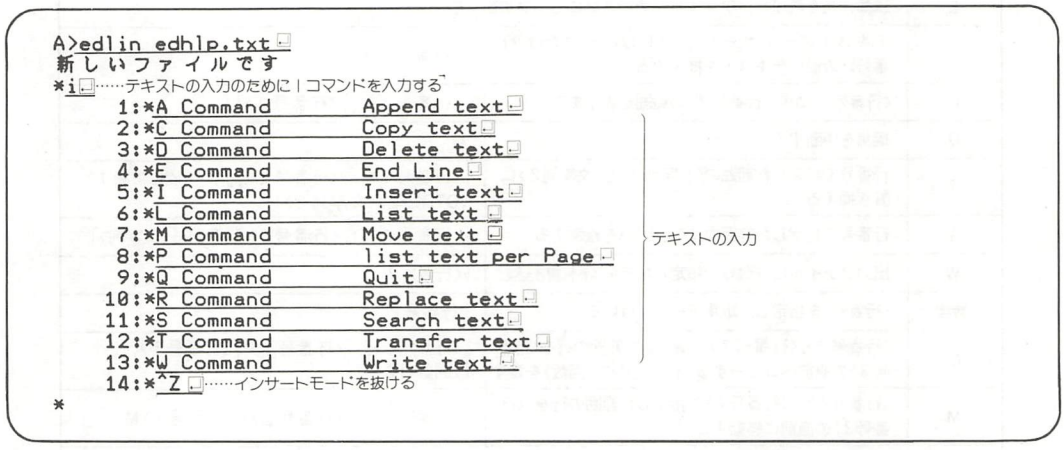


Fig. 2．8　テキストの作成例

テキストの表示(List)

Fig. 2.8 で入力したテキストでは，4行目を〝End Edit〟とすべきところが，ミスタイプで〝End Line〟になっています．とりあえずこの行を修正して，確認のためにLコマンドでテキストを表示させてみます．

特定の行を編集する場合には，行番号を指定します．指定された行の内容はテンプレートに収められていますから，Fig. 2.9 で修正しているようにテンプレート機能を用いれば，入力し直すよりも効率よく処理できます．

```
*4↵……第4行を指定する
     4:*E Command        End Line
     4:*E Command        End Edit CU L E d i t ↵
*1↵……テキストの表示(Lコマンドの実行)
     1: A Command        Append text
     2: C Command        Copy text
     3: D Command        Delete text
     4:*E Command        End Edit
     5: I Command        Insert text
     6: L Command        List text
     7: M Command        Move text
     8: P Command        list text per Page
     9: Q Command        Quit
    10: R Command        Replace text
    11: S Command        Search text
    12: T Command        Transfer text
    13: W Command        Write text
*
    テキストの修正のために第4行を指定したため，現在行は第4行になっている
    〝4：〞の右側にある〝*〞は現在行を意味している
```

Fig. 2．9　行番号の指定による修正

Lコマンドは，Fig. 2.9で実行したように単に〝L〞のみを入力しても，現在行を中心として23行のテキストであればすべてのテキストを表示することができますが，テキストの一部を表示するようなときは，行番号で範囲を指定して入力を行います．Fig. 2.10では，Lコマンドの行番号の指定に〝#〞記号が使用されていますが，この記号はテキストの〝最終行の行番号+1〞を意味しています．

```
*1,3↵……第1～第3行を表示する
     1:*A Command        Append text
     2: C Command        Copy text
     3: D Command        Delete text
*4,#1↵……第4行からテキストの最後までを〝#〞記号を使って表示する
     4: E Command        End Edit
     5: I Command        Insert text
     6: L Command        List text
     7: M Command        Move text
     8: P Command        list text per Page
     9: Q Command        Quit
    10: R Command        Replace text
    11: S Command        Search text
    12: T Command        Transfer text
    13: W Command        Write text
*
```

Fig. 2．10　Listコマンドの実行例

文字列の検索と置換(Search, Replace)

ラインエディタで編集作業をする場合，特定の行番号を指定してテキストの編集を行うようなことはあまり行われません．ここで扱うような小さなテキストであれば，ひと目でテキストの全体を見渡すことができますが，高級言語やアセンブラのソースリストのような大きなテキストは，一度に把握できる範囲が限られてしまうために，行番号を指定して編集を行うには不向きです．こういったテキストを効率よく編集するためのコマンドがR，Sコマンドです．

Sコマンドは，指定した文字列をテキストの中から検索するものですが，さきに入力したテキストの中から〝text〞を見つけ出す例をFig. 2.11に示します．

```
*1,#stext↵……テキストの先頭から最後の行までの範囲で文字列 "text" を検索する
    1:*A Command          Append text……最初に "text" が発見された行で検索は中止される
*
```

Fig. 2.11　Searchコマンドの実行例

Fig. 2.11 の例では，テキスト中の最初の検索文字列を見つけ出した行でSコマンドは終了していますが，テキスト全体にわたって指定した文字列を含む行を検索したい場合には Fig. 2.12 に示すように〝?〟オプションを用います．

〝よろしいですか　〈Y／N〉?〟でリターンキーか〝Y〟を入力すると，その行を着目行として検索を終了します．

```
*1,#?stext↵……"?" オプションを付けてテキスト全体にわたって文字列の検索を行う
    1:*A Command          Append text
よろしいですか 〈Y/N〉? n↵……"n" を入力すると次の "text" を含む行を検索する
    2: C Command          Copy text
よろしいですか 〈Y/N〉? n↵
    3: D Command          Delete text
よろしいですか 〈Y/N〉? y↵……"y" を入力すると検索を終了する
*
```

Fig. 2.12　〝?〟オプションを用いたSearchコマンドの実行例

Rコマンドは，文字列の置換えに使用されるコマンドですが，Fig. 2.13 では〝text〟をすべて〝TEXT〟に置き換えています．もちろん〝?〟オプションを付けて，置換えを行うか否かの確認をとりながらの処理もできます．

```
*1,#rtext^ZTEXT↵……テキスト全体にわたって "text" を "TEXT" に置き換える
    1:*A Command          Append TEXT
    2: C Command          Copy TEXT
    3: D Command          Delete TEXT
    5: I Command          Insert TEXT
    6: L Command          List TEXT
    7: M Command          Move TEXT
    8: P Command          list TEXT per Page
   10: R Command          Replace TEXT
   11: S Command          Search TEXT
   12: T Command          Transfer TEXT
   13: W Command          Write TEXT
*
```

置換の対象となった行のみが表示される

Fig. 2.13　Replaceコマンドの実行例

SコマンドやRコマンドは，プログラムリストの変数名やラベルの付換えに頻繁に使用されますが，文書ファイルなどでは，Fig. 2.14 に示すように，長い文字列や漢字などを簡単な記号で入力しておき，後で変換するようなことができるでしょう．

```
*1,#l↵……以下のテキストでは"東京都"を"T．"で代用して入力している
    1:*T.        三鷹市　牟礼 XXXXXXXXXXXXXXXXXXXXXXX
    2: T.        港区　南青山 XXXXXXXXXXXXXXXXXXXXXXX
    3: T.        武蔵野市　吉祥寺 XXXXXXXXXXXXXXXXXXXX
    4: T.        杉並区　久我山 XXXXXXXXXXXXXXXXXXXXX
    5: 茨城県    取手市 XXXXXXXXXXXXXXXXXX
    6: 千葉県    我孫子市 XXXXXXXXXXXXXXXXXXXXX
    7: 埼玉県    浦和市 XXXXXXXXXXXXXXXXXX
    8: 神奈川県  横浜市 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    9: 神奈川県  川崎市 XXXXXXXXXXXXXXXXXXXXXXXXXX
   10: T.        荒川区　東日暮里 XXXXXXXXXXXXXXXXXXXX
   11: T.        文京区　小日向 XXXXXXXXXXXXXXXXXXXXXX
*1,#rT.^Z東京都↵……Rコマンドで"T．"を"東京都"に置換する
    1:*東京都    三鷹市　牟礼 XXXXXXXXXXXXXXXXXXXXXXX
    2: 東京都    港区　南青山 XXXXXXXXXXXXXXXXXXXXXXX
    3: 東京都    武蔵野市　吉祥寺 XXXXXXXXXXXXXXXXXXXX
    4: 東京都    杉並区　久我山 XXXXXXXXXXXXXXXXXXXXX
   10: 東京都    荒川区　東日暮里 XXXXXXXXXXXXXXXXXXXX
   11: 東京都    文京区　小日向 XXXXXXXXXXXXXXXXXXXXXX
*1,#l↵
    1: 東京都    三鷹市　牟礼 XXXXXXXXXXXXXXXXXXXXXXX
    2: 東京都    港区　南青山 XXXXXXXXXXXXXXXXXXXXXXX
    3: 東京都    武蔵野市　吉祥寺 XXXXXXXXXXXXXXXXXXXX
    4: 東京都    杉並区　久我山 XXXXXXXXXXXXXXXXXXXXX
    5: 茨城県    取手市 XXXXXXXXXXXXXXXXXX
    6: 千葉県    我孫子市 XXXXXXXXXXXXXXXXXXXXX
    7: 埼玉県    浦和市 XXXXXXXXXXXXXXXXXX
    8: 神奈川県  横浜市 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    9: 神奈川県  川崎市 XXXXXXXXXXXXXXXXXXXXXXXXXX
   10:*東京都    荒川区　東日暮里 XXXXXXXXXXXXXXXXXXXX
   11: 東京都    文京区　小日向 XXXXXXXXXXXXXXXXXXXXXX
*
```

置換された行のみが表示される

Fig. 2．14　文字列の置換操作

テキストの挿入，移動，コピー，ページ出力，削除(Insert, Move, Copy, Page, Delete)

Iコマンドは，新規のファイルを作成する前に用いるだけでなく，任意の行にテキストを挿入する場合にも使用します．Fig. 2.15 は，さきほどのテキストの最後にテキストを追加したものです．

```
*#i↵……テキストの最終行＋1行目からテキストを挿入する
   14:*EDLIN has following commands.↵
   15:*↵
   16:*^Z↵……インサートモードを抜ける
*1,#l↵
    1: A Command        Append text
    2: C Command        Copy text
    3: D Command        Delete text
    4: E Command        End Edit
    5: I Command        Insert text
    6: L Command        List text
    7: M Command        Move text
    8: P Command        list text per Page
    9: Q Command        Quit
   10: R Command        Replace text
   11: S Command        Search text
   12: T Command        Transfer text
   13: W Command        Write text
   14: EDLIN has following commands.
   15:
*e↵
```

テキストの最後に挿入された2行

Fig. 2．15　Insertコマンドの実行例

Mコマンドは，指定した範囲のテキストを任意の位置に移動するコマンドですが，Fig. 2.16で最初に入力したテキストと追加したテキストの部分を入れ換えたもの，つまり14〜15の範囲のテキストを先頭行に移動させます．

```
*14,15,1m↵……Fig.2.15で追加したテキストをテキストの先頭へ移動する
*1,#↵
    1:*EDLIN has following commands.   } 移動されたテキスト
    2:
    3: A Command        Append text
    4: C Command        Copy text
    5: D Command        Delete text
    6: E Command        End Edit
    7: I Command        Insert text
    8: L Command        List text
    9: M Command        Move text
   10: P Command        list text per Page
   11: Q Command        Quit
   12: R Command        Replace text
   13: S Command        Search text
   14: T Command        Transfer text
   15: W Command        Write text
*
```

Fig. 2.16　Moveコマンドの実行例

プログラムリストの編集で，類似したサブルーチンをコピーして一部変更するとか，同じ命令が繰り返し使われるような場合に，Cコマンドを用います．

Fig. 2.17では，今まで入力したテキストを最下行にそっくりコピーしています．

```
*1,15,#c↵……第1〜15行のテキストの最後にコピーする
*1,#↵
    1: EDLIN has following commands.
    2:
    3: A Command        Append text
    4: C Command        Copy text
    5: D Command        Delete text
    6: E Command        End Edit
    7: I Command        Insert text
    8: L Command        List text
    9: M Command        Move text
   10: P Command        list text per Page
   11: Q Command        Quit
   12: R Command        Replace text
   13: S Command        Search text
   14: T Command        Transfer text
   15: W Command        Write text
   16: EDLIN has following commands.
   17:
   18: A Command        Append text
   19: C Command        Copy text
   20: D Command        Delete text
   21: E Command        End Edit
   22: I Command        Insert text
   23: L Command        List text
   24: M Command        Move text
   25: P Command        list text per Page
   26: Q Command        Quit
   27: R Command        Replace text
   28: S Command        Search text
   29: T Command        Transfer text
   30: W Command        Write text
*
```

コピーされたテキスト

Fig. 2.17　Copyコマンドの実行例

ここで示したような小さなテキストでは，テキストを1画面ごとに順次出力するPコマンドを使用することはありませんが，何百行にもわたる大きなテキストを扱うような場合，全体を見渡して編集する領域を決定します．Fig. 2.18はPコマンドの実行例です．なお，画面に一度に表示されるのは23行です．

```
*p↵……行番号を指定しないでPコマンドを実行すると現在行の次の行から23行表示される
    1: EDLIN has following commands.
    2:
    3: A Command        Append text
    4: C Command        Copy text
    5: D Command        Delete text
    6: E Command        End Edit
    7: I Command        Insert text
    8: L Command        List text
    9: M Command        Move text
   10: P Command        list text per Page
   11: Q Command        Quit
   12: R Command        Replace text
   13: S Command        Search text
   14: T Command        Transfer text
   15: W Command        Write text
   16: EDLIN has following commands.
   17:
   18: A Command        Append text
   19: C Command        Copy text
   20: D Command        Delete text
   21: E Command        End Edit
   22: I Command        Insert text
   23:*L Command        List text
*p↵……現在行は表示されたテキストの最後の行に移動する
……前のテキストの最終行の次の行から表示される
   24: M Command        Move text
   25: P Command        list text per Page
   26: Q Command        Quit
   27: R Command        Replace text
   28: S Command        Search text
   29: T Command        Transfer text
   30:*W Command        Write text
*
```

Fig. 2.18　Pageコマンドの実行例

Dコマンドは，指定範囲のテキストの削除を行うものです．Cコマンドで追加したテキストを削除して，Eコマンドでテキストをいったんディスクにセーブしておきます(Fig. 2.19)．

```
*16,#d↵……第16行からテキストの最後までを削除する
*1,#l↵
    1: EDLIN has following commands.
    2:
    3: A Command        Append text
    4: C Command        Copy text
    5: D Command        Delete text
    6: E Command        End Edit
    7: I Command        Insert text
    8: L Command        List text
    9: M Command        Move text
   10: P Command        list text per Page
   11: Q Command        Quit
   12: R Command        Replace text
   13: S Command        Search text
   14: T Command        Transfer text
   15: W Command        Write text
*e↵……EDLINの終了．テキストがディスクに書き込まれる

A>■
```

第16行以降のテキストが削除されている

Fig. 2.19　Deleteコマンドの実行とテキストのセーブ(ENDコマンド)

○既存のテキストファイルの編集

プログラム開発のプロセスでは，プログラムを先頭から書くということはせずに，通常は機能的なまとまりごとにモジュールとして作成し，後で各モジュールを結合するという方法をとります．また，こうして作られたソース・プログラムは，メモリの少ないシステムではそのすべてをメモリ上に置くことができないために，ディスクとのデータのやり取りを行いながら，テキストの一部をメモリにおいて編集する場合があります．

このような大きなテキストを編集する場合に，ディスク上のファイルとメモリ内のテキストを結合したり，ディスクを外部メモリとして利用する場合に用いられるのがT，W，Aの各コマンドです．

テキストの結合(Transfer)

Tコマンドは，最初に読み出したテキストに対してさらに別のファイルのテキストを追加するような場合に使用されます．Fig. 2.20では，さきほどディスクに書き込んだファイル〝EDHLP．TXT〟をEDLINで読み出して，さらにTコマンドで〝EDHLP．TXT〟を結合した例を示します．

```
A>edlin edhlp.txt ↵……Fig.2.19でディスクにセーブしたテキストを，再度EDLINで読み込む
ファイルを終りまで読込みました
*10tedhlp.txt ↵……Tコマンドによって，第10行以降にもう一度〝EDHLP．TXT〟を読み込み，
*1,#↵                もとのテキストに結合する
     1: EDLIN has following commands.
     2:
     3: A Command        Append text
     4: C Command        Copy text
     5: D Command        Delete text
     6: E Command        End Edit
     7: I Command        Insert text
     8: L Command        List text
     9: M Command        Move text
    10:*EDLIN has following commands.
    11:
    12: A Command        Append text
    13: C Command        Copy text
    14: D Command        Delete text
    15: E Command        End Edit
    16: I Command        Insert text
    17: L Command        List text
    18: M Command        Move text
    19: P Command        list text per Page
    20: Q Command        Quit
    21: R Command        Replace text
    22: S Command        Search text
    23: T Command        Transfer text
    24: W Command        Write text
    25: P Command        list text per Page
    26: Q Command        Quit
    27: R Command        Replace text
    28: S Command        Search text
    29: T Command        Transfer text
    30: W Command        Write text
*
```

Tコマンドによって結合されたテキスト

Fig. 2.20　Transferコマンドの実行例

テキストの読出し／書込み(Append, Write)

A，Wの2つのコマンドは，通常ペアで利用され，ディスクとメモリとの間でテキストのやり取りを行いながら編集を行うためのコマンドです．まずA，Wコマンドの準備のために，Cコマンドでメモリいっぱいまでテキストを作成してディスクにファイルを作成します．ファイル名は"BIG．TXT"で，テキストの最終行を判別できるようにアルファベットを書き込んでおきます(Fig. 2.21)．

```
A>edlin big.txt↵……ファイル名 "BIG．TXT" を新規ファイルとして登録する
新しいファイルです
*i↵……インサートモードにはいる
     1:*00000000000000000000↵
     2:*11111111111111111111↵
     3:*22222222222222222222↵
     4:*33333333333333333333↵
     5:*44444444444444444444↵
     6:*55555555555555555555↵
     7:*66666666666666666666↵
     8:*77777777777777777777↵
     9:*88888888888888888888↵
    10:*99999999999999999999↵
    11:*^Z↵
                                  } 新規テキストの作成
*1,10,#,254c↵……Cコマンドによって第1～第10行までのテキストを254回繰り返し，テキストの最後に追加する
*i#↵
  2551:*ABCDEFGHIJKLMNOPQRSTUVWXYZ↵……最終行の目印として2551行にアルファベットを書き込む
  2552:*^Z↵
*e↵……EDLINを終了し，"BIG．TXT" をディスクに書き込む

A>dir big.*↵……"BIG．TXT" のファイルの存在を確認する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

BIG      TXT    56129  84-04-24  22:04
       1 個のファイルがあります
  770048 バイトが使用可能です

A>■
```

Fig. 2.21　BIG．TXTの準備

次に，EDLINで"BIG．TXT"を読出し，着目行を最後に移してテキストを表示します(Fig. 2.22)．新規に作成したファイルが2551行であったのに，メモリには1914行までしか読み込まれていません．そこで，残りのテキストを読み込むために1～1000行をWコマンドでディスクに書き込み，メモリに空き領域を作った上でAコマンドで残りのテキストをメモリに読み込みます．

```
A>edlin big.txt↵…… "BIG．TXT" を読み込む．ファイルをすべて読み込んでいないためにEDLINからのメッセージ「ファイルを終り
                    まで読込みました」が表示されない
*#↵……現在行のポインタをテキストの最後に移動する
*l↵……テキストの最後部から11行を表示する．Fig2.21で作成したファイルが2551行あったのに対し，現在メモリに読み込まれているのは1914行
                                                                    までしかない
  1904: 33333333333333333333
  1905: 44444444444444444444
  1906: 55555555555555555555
  1907: 66666666666666666666
  1908: 77777777777777777777
  1909: 88888888888888888888
  1910: 99999999999999999999
  1911: 00000000000000000000
  1912: 11111111111111111111
  1913: 22222222222222222222
  1914: 33333333333333333333
```

```
*1000w ……第1～第1000行をディスクに書き込み，メモリに空き領域を作る
*1000a ……"BIG．TXT" の残りの1000行をメモリに読み込む
ファイルを終りまで読込みました ……ディスクに残っていたファイルをすべて読み込んだので，EDLINからメッセージが返される
*# ……現在行のポインタをテキストの最後に移動する
*1
  1541: 00000000000000000000
  1542: 11111111111111111111
  1543: 22222222222222222222
  1544: 33333333333333333333
  1545: 44444444444444444444
  1546: 55555555555555555555
  1547: 66666666666666666666
  1548: 77777777777777777777
  1549: 88888888888888888888
  1550: 99999999999999999999
  1551: ABCDEFGHIJKLMNOPQRSTUVWXYZ ……テキストの最後を示すアルファベットの行が読み込まれている
*e ……EDLINを終了し，テキストをディスクに書き込む

A>■
```

Fig. 2.22　W，Aコマンドの実行例

バックアップファイルの確認

ここで，今作成した〝BIG．TXT〟ファイルをDIRコマンドで表示してみます．

```
A>dir big.*

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

BIG      BAK    56129  84-04-24   22:04 ……既存のテキストの編集後には拡張子〝．BAK〟の付いたファイルが作成
BIG      TXT    56129  84-04-24   22:08   される．このファイルはEDLINで編集する前のファイルをバックアップ
        2 個のファイルがあります          として保存されている
   713728 バイトが使用可能です

A>■
```

Fig. 2.23　バックアップファイルの確認

Fig. 2.23に示すように，〝BIG．TXT〟のほかにもう1つ〝BIG．BAK〟というファイルが作られているのがわかります．これは，Eコマンドで編集の終了したファイルをディスクに書き込んだ際に，オリジナルファイルの拡張子を〝．BAK〟に変更しているものです(Fig. 2.24)．

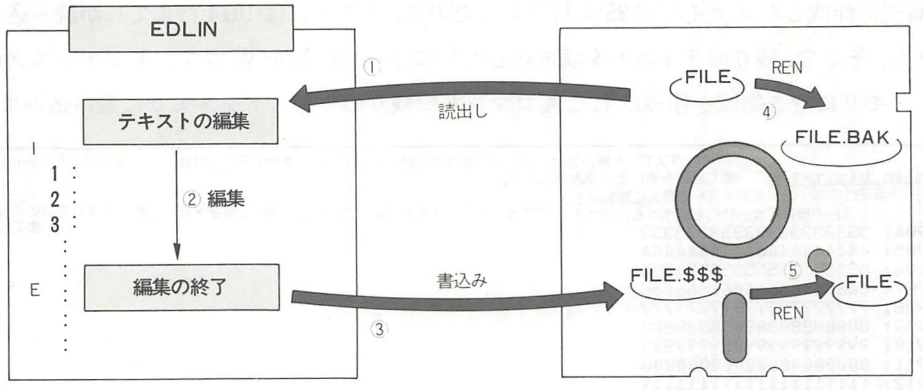


Fig. 2.24　EDLINが〝．BAK〟を作る手順

EDLIN を使用する場合に注意しなければならないのは，カレントディスクの残り容量です．E コマンドを実行した際に，ディスクの残り容量がファイル全体を書き込むのに十分な空き領域がないと，Fig. 2.25 に示すようなエラーメッセージが表示されてファイルの書込みは中止され，そのとき書込みに失敗したファイル，つまり編集を行っていたファイルは失われてしまいます．

```
A>edlin address ……Fig. 2.14で作成したファイルを読み込む．ファイル名は"ADDRESS"
ファイルを終りまで読込みました
*1,11,#,10c
*e
ディスクがいっぱいです．書込みが完了しませんでした
A>■
```

ファイルを書き込んだ時点でディスクにファイルを書き込むだけの空き領域がないと上に示すようなエラーメッセージが表示され，ディスクへの書込みが中止される

Fig. 2.25　ディスクフルのエラー

このときのディスクの内容を見てみましょう(Fig. 2.26)．

```
A>dir address.*

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

                    ファイルの大きさは0バイト．書込みは失敗
ADDRESS  $$$          0  84-04-24  22:44 ……テンポラリファイル
ADDRESS             480  84-04-24  22:42 ……編集前のファイル
       2 個のファイルがあります
    2048 バイトが使用可能です

A>■
```

Fig. 2.26　ディスクフルエラーの発生したディスク

テンポラリファイル(〝. $$$〞)と編集前のファイルは残っていますが，テンポラリファイルの大きさは 0 バイトになっており，書込みに失敗したことを物語っています．

このような事故を完全に防ぐことは不可能でしょうが，最低限次の点には注意する必要があるでしょう．

(1) CHKDSK コマンドでディスクの残り容量を確認する

(2)既存のファイルを編集するのであれば，DIR コマンドによって現在のファイルの大きさを調べ，ディスクの残り容量より大きければほかのディスクにコピーしてから編集を行う

2.2.4　ファイルの内容表示(TYPE)

TYPE コマンドは，EDLIN やほかの MS-DOS で動くエディタなどで作成したアスキーファイルの内容を表示するものです．PC-9800 シリーズ用の MS-DOS には〝README．DOC〞〝COPYA．DOC〞などのドキュメントファイルが含まれていますが，これらのファイルの内容も TYPE コマンドで見ることができます．

Fig. 2.27は前項で作成した〝EDHLP．TXT〟を表示したもので，Fig. 2.28は，〝README．DOC〟の内容の一部です．

```
A>type edhlp.txt ↵……〝EDHLP．TXT〟のファイルの内容をタイプアウトする
EDLIN has following commands.

A Command       Append text
C Command       Copy text
D Command       Delete text
E Command       End Edit
I Command       Insert text
L Command       List text
M Command       Move text
P Command       list text per Page
Q Command       Quit
R Command       Replace text
S Command       Search text
T Command       Transfer text
W Command       Write text

A>■
```

タイプアウトされたファイルの内容

Fig. 2.27　TYPEコマンドの実行例①

```
A>type readme.doc ↵……PC-9800シリーズのMS-DOSに含まれているドキュメントファイル

    このドキュメントでは、ＭＳ－ＤＯＳ　２．０の補足事項、注意／制限事項につ
いて解説します．

（１）　辞書ファイル

                                ライブ上のルートディレクトリになければ、正
```

Fig. 2.28　TYPEコマンドの実行例②

○バイナリファイルの表示

TYPEコマンドはアスキーファイルの内容を表示すると述べましたが，確認のために〝DEBUG．COM〟(拡張子〝．COM〟の付いたファイルは実行可能なバイナリファイル)をTYPEコマンドで表示してみます．Fig. 2.29に示すように意味不明の内容が表示されます．

```
A>type debug.com ↵……〝DEBUG．COM〟をタイプアウトしてみる
シ ,「S.ﾇﾍｵﾗこ2徳寅0ﾍｻｺﾍ■ ■ ■ ﾐIﾀｸｲ卯ソー,ﾍ!ﾀｫ■ZIQW」d.ｸ 」X.」T.」b.尺 後ｺ  I
〝DEBUG．COM〟はバイナリファイルなので，TYPEコマンドではファイルの内容を表示することはできない
```

Fig. 2.29　バイナリファイルの表示

2.2.5　ファイルの削除(DEL, ERASE)

2.2.3ではEDLINによるファイルの作成，編集について解説しましたが，ファイルを作成する手順は複雑ではないにしろ，エディタの操作に慣れるまでは多少手間のかかる作業です．しかしいったん作成したファイルを消去するのは，DELコマンドでいとも簡単に行えます．この操作の

単純さが，ときとして必要なファイルを削除するという思わぬ事故を引き起こしますから，DELコマンドを使用する際は十分な注意が必要です．

Fig. 2.30 では，まずDIRコマンドでカレント・ドライブのディスクの内容を確認して，さきほどA，Wコマンドの実験で作った〝BIG．TXT〟を削除します．

```
A>dir /w ……DELコマンドを実行する前に，ディスクの内容を確認する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

COMMAND  COM    KNJDIC   SYS    CHKDSK   COM    DEBUG    COM    EDLIN    COM
MORE     COM    RECOVER  COM    SYS      COM    CREF     EXE    EXE2BIN  EXE
FC       EXE    FIND     EXE    LIB      EXE    LINK     EXE    MASM     EXE
SORT     EXE    ASSIGN   COM    COPY2    COM    COPYA    COM    DICM     COM
DISKCOPY COM    DUMP     COM    FORMAT   COM    KEY      COM    PRINT    COM
SPEED    COM    SWITCH   COM    USKCGM   COM    COPYA    DOC    README   DOC
EDHLP    BAK    ADDRESS  $$$    EDHLP    TXT    BIG      BAK    BIG      TXT
ADDRESS                                                          削除するファイル
        36 個のファイルがあります
   714752 バイトが使用可能です

A>del big.txt ……〝BIG．TXT〟を削除する

A>dir /w

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

COMMAND  COM    KNJDIC   SYS    CHKDSK   COM    DEBUG    COM    EDLIN    COM
MORE     COM    RECOVER  COM    SYS      COM    CREF     EXE    EXE2BIN  EXE
FC       EXE    FIND     EXE    LIB      EXE    LINK     EXE    MASM     EXE
SORT     EXE    ASSIGN   COM    COPY2    COM    COPYA    COM    DICM     COM
DISKCOPY COM    DUMP     COM    FORMAT   COM    KEY      COM    PRINT    COM
SPEED    COM    SWITCH   COM    USKCGM   COM    COPYA    DOC    README   DOC
EDHLP    BAK    ADDRESS  $$$    EDHLP    TXT    BIG      BAK    ADDRESS
        35 個のファイルがあります ……〝BIG．TXT〟が削除されている
   771072 バイトが使用可能です
```

Fig. 2.30　DELコマンドの実行例

DELコマンドではワイルドカードが使用できるので，条件に見合ったファイルをまとめて削除することができます．Fig. 2.31 では，拡張子〝．EXE〟の付いたファイルをすべて削除しています．

```
A>dir *.exe /w ……拡張子〝．EXE〟のファイルの一覧を表示する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

CREF     EXE    EXE2BIN  EXE    FC       EXE    FIND     EXE    LIB      EXE
LINK     EXE    MASM     EXE    SORT     EXE
        8 個のファイルがあります
   771072 バイトが使用可能です

A>del *.exe ……拡張子〝．EXE〟のファイルをすべて削除する

A>dir *.exe /w ……もう一度拡張子〝．EXE〟のファイル名一覧を表示させようとしても，エラーメッセージが表示される

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

ファイルが見つかりません

A>■
```

Fig. 2.31　ワイルドカードを用いたDELコマンドの実行例

ワイルドカードを用いてすべてのファイルを削除することもできます．しかし，このコマンドの実行は復旧が不可能な結果となるため，Fig. 2.32 のように，システム側でも一度実行するか否かの確認をとります．ここで"Y"あるいはリターンキーを押せば，すべてのファイルは削除されます．

```
A>del *.*↵……すべてのファイルを削除する
よろしいですか <Y/N>? y ……"y"あるいはリターンキーを押すと，すべてのファイルが削除される
  └システム側((MS-DOS)からコマンドの実行の確認がとられる
A>dir↵

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

ファイルが見つかりません……ファイル名一覧を表示するコマンドを実行しても，ファイルが存在しないためにエラーメッセージが
                            表示される
A>■
```

Fig. 2 .32　すべてのファイルの削除

2.2.6　ファイルのコピー(COPY)

COPY コマンドは，MS-DOS のコマンドの中でも最も多くの機能を備えたものであり，ファイル操作では頻繁に使用されるコマンドの１つです．

COPY コマンドの機能は，大きく次の３つに分けることができます．

(1)ファイルのコピー(複製)
(2)ファイルの結合
(3)データの転送(周辺装置)

このうち，ここでは(1)，(2)の機能について解説を行い，(3)については後述する標準入出力やデバイスファイルの概念と併せて解説することにします．

○ファイルのコピー

COPY コマンドは，ファイルの保護，プログラムやデータの共有，プログラム・モジュールの結合などの目的に使用されますが，まず単純な例としてカレント・ドライブ(ここではドライブＡを指定)のディスク上の１つのファイル"EDLIN．COM"をドライブＢのディスクにコピーしてみます(Fig. 2.33)．

```
A>copy edlin.com b:↵……ドライブAのディスクにある "EDLIN．COM" をドライブBのディスクにコピーする
        1 個のファイルをコピーしました

A>dir b:↵……ドライブBの内容を確認する

 ドライブ B: のボリュームラベルは PC9800_SYS
 ディレクトリは B:¥

EDLIN    COM      9124  83-09-17  12:57  ……ドライブBのディスクに "EDLIN．COM" がコピーされている
        1 個のファイルがあります
  1181696 バイトが使用可能です

A>■
```

Fig. 2．33　COPYコマンドの実行例

COPY コマンドではワイルドカードが利用できますので，特定の条件にマッチしたファイルを一度にコピーすることもできます．Fig. 2.34 では，拡張子に〝．COM〟の付いたファイルをドライブＢにコピーしています．

```
A>copy *.com b:↵……ワイルドカードを用いてドライブAのディスクにある拡張子 〝．COM〟 のファイルをすべてドライブBにコピーする
COMMAND.COM
CHKDSK.COM
DEBUG.COM
EDLIN.COM
MORE.COM
RECOVER.COM
SYS.COM
ASSIGN.COM
COPY2.COM
COPYA.COM          } コピーされたファイルがスクリーンに出力される
DICM.COM
DISKCOPY.COM
DUMP.COM
FORMAT.COM
KEY.COM
PRINT.COM
SPEED.COM
SWITCH.COM
USKCGM.COM
       19 個のファイルをコピーしました

A>dir b:*.com /w↵……ドライブBのディスクの内容を確認する

 ドライブ B: のボリュームラベルは PC9800_SYS
 ディレクトリは B:¥

EDLIN     COM    COMMAND  COM    CHKDSK   COM    DEBUG    COM    MORE     COM
RECOVER   COM    SYS      COM    ASSIGN   COM    COPY2    COM    COPYA    COM
DICM      COM    DISKCOPY COM    DUMP     COM    FORMAT   COM    KEY      COM
PRINT     COM    SPEED    COM    SWITCH   COM    USKCGM   COM
       19 個のファイルがあります …… COPYコマンドの実行によってコピーされたファイル名がすべて表示される
  1077248 バイトが使用可能です

A>■
```

Fig. 2．34　ワイルドカードを用いたCOPYコマンドの実行例

○ファイルの結合

プログラムがモジュールごとに開発されており，単に結合するだけであればEDLINのTコマンドで何度もファイルを読み込むよりもCOPYコマンドのファイル結合機能を用いれば簡単な操作で処理することができます．Fig. 2.35 の例は，3つのプログラム・モジュールを結合して

"PROG1．BAS" というファイルにしたものです．

```
A>dir mod*.* /w ……"MOD" ではじまるファイルをすべて表示する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

MOD1      BAS     MOD2      BAS     MOD3      BAS
        3 個のファイルがあります
  1074176 バイトが使用可能です

A>type mod1.bas
100 ' main
110 GOSUB 200
120 X=30 : Y=10 : GOSUB 300
130 PRINT "How are you ?"
140 END

A>type mod2.bas                    各ファイルの内容を表示する
200 ' cls
210 PRINT CHR$(&H1B)+"[2J";
220 RETURN

A>type mod3.bas
300 ' locate
310 PRINT CHR$(&H1B)+"["+MID$(STR$(Y),2)+";"+MID$(STR$(X),2)+"H";
320 RETURN
                                   結合したファイルのファイル名
A>copy mod1.bas+mod2.bas+mod3.bas prog1.bas
MOD1.BAS
MOD2.BAS  } 3つのファイルを結合する
MOD3.BAS
        1 個のファイルをコピーしました

A>type prog1.bas ……"PROG1．BAS" の内容をタイプアウトする．3つのファイルが結合されていることが確認できる
100 ' main
110 GOSUB 200
120 X=30 : Y=10 : GOSUB 300
130 PRINT "How are you ?"
140 END
200 ' cls
210 PRINT CHR$(&H1B)+"[2J";
220 RETURN
300 ' locate
310 PRINT CHR$(&H1B)+"["+MID$(STR$(Y),2)+";"+MID$(STR$(X),2)+"H";
320 RETURN

A>■
```

Fig. 2．35　ファイルの結合

○データファイルのバックアップ

データベースや作表言語で作り上げたデータは，通常データディスクとしてまとめられ，一定期間ごとにバックアップをとって保存されます．このような場合，DISKCOPY コマンドでは，ディスクのすべてのプログラムがコピーされてしまうので，ディスクの空き領域がむだに使われてしまいます．システム・プログラムを除いたデータファイルだけをコピーするような場合には COPY コマンドとワイルドカードを組み合せて Fig. 2.36 のように実行するとよいでしょう．

逆にいえば，COPY コマンドでは，ディスクをまるごとコピーすることはできません(第II部1章 SYS コマンド参照)．

```
A>dir /w ↵……ドライブAのディスクの内容を確認する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

COMMAND  COM     SAMP      TXT     TEST      TXT     LANGUAGE TXT     OPR       TXT
       5 個のファイルがあります
 1167360 バイトが使用可能です

A>copy *.* b: ↵……ドライブAのディスクに含まれるすべてのファイルをドライブBのディスクにコピーする
COMMAND.COM
SAMP.TXT
TEST.TXT
LANGUAGE.TXT
OPR.TXT
       5 個のファイルをコピーしました

A>■
```

Fig. 2.36　すべてのファイルのコピー

2.2.7　ファイル名の変更(REN コマンド)

REN コマンドは，ディスク上のファイルの数が増えてくると，関係のあるファイルに統一的なファイル名を付け換えたり，変更を加えた古い内容のファイルを別のファイル名に付け換えるなど，ファイルの整理にはよく用いられるコマンドです．

Fig. 2.37 では，拡張子に〝. TXT〞の付いたファイルを，すべて〝. PRN〞に変更しています．

```
A>dir *.txt /w ↵……拡張子が〝. TXT〞のファイルをすべて表示する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

SAMP     TXT   TEST     TXT   LANGUAGE TXT   OPR      TXT
       4 個のファイルがあります
 1167360 バイトが使用可能です

A>ren *.txt *.prn ↵……拡張子が〝. TXT〞のついたファイルをすべて〝. PRN〞に変更する

A>dir *.prn /w ↵

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

SAMP     PRN   TEST     PRN   LANGUAGE PRN   OPR      PRN
       4 個のファイルがあります
 1167360 バイトが使用可能です

A>■
```

Fig. 2.37　ワイルドカードを用いたファイル名の変更

REN コマンドを使用する上で注意しなければならないのは，同じディスク上にすでに存在するファイル名を指定した場合はエラーとなることです(Fig. 2.38)．

```
A>dir *.txt /w ↵……拡張子 ". TXT" のファイルをすべて出力する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

SAMP     TXT    TEST     TXT    LANGUAGE TXT    OPR      TXT
        4 個のファイルがあります
1167360 バイトが使用可能です

A>ren samp.txt test.txt ↵……すでに存在するファイル名に名前変更を行うとエラーが発生する
ファイル名が重複しているか，または ファイルが見つかりません

A>■
```

Fig. 2.38　RENコマンドのエラー発生

2.3 ディスクとファイル管理

ユーザー側で扱うファイルは，ファイル名の付けられた論理的に意味のあるデータの集合ですが，ディスク上では物理的なセクタ単位のデータとして記録されています．この両者の構造には大きなギャップがあるために，ファイルを管理する上ではディレクトリと呼ばれるファイルの管理情報を記録しておく領域を設定し，ユーザーの捉えているファイルのイメージを物理的なディスクの位置に結び付けています．ディレクトリにはファイルの管理情報として，ファイル名，ファイルサイズ，ディスク上の位置情報などが収められており，この情報をもとにMS-DOSはファイルの管理を行っています(第III部5章参照)．

ディレクトリによるファイルの管理というのは，何もMS-DOSに限ったことではなく，他のDOSやDISK BASICにおいても基本的には同じ処理を行っています．ただしMS-DOSは，ファイル管理の方法においてもUNIXの影響を強く受けているため，階層的に複数のディレクトリを持っているのです．

2.3.1 階層的ディレクトリ構造

階層的ディレクトリ構造とは，Fig. 2.39に示すように下方に成長する木の形を成しており(木構造)，いちばん上が根(ルート)に，ディレクトリは枝に，そして個々のファイルは葉にたとえられています．この図では，BIN，TOOLS，TXTがディレクトリであり，"COMMAND．COM"がファイルになります．いちばん上の根の部分は，その名のとおりルート・ディレクトリと呼ばれ，他のすべてのディレクトリはサブ・ディレクトリと呼ばれます．

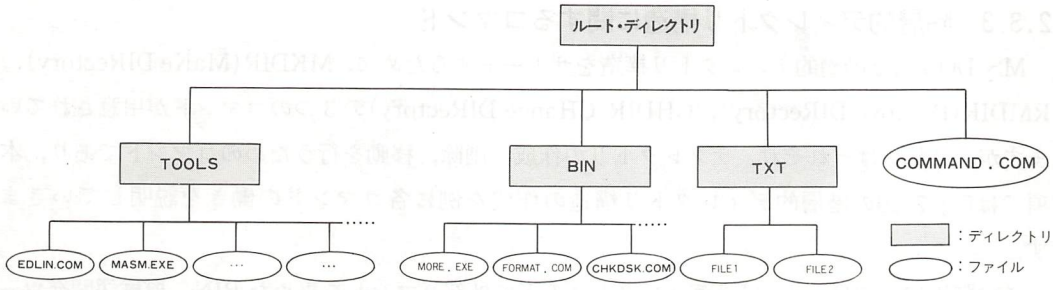


Fig. 2 .39　階層的ディレクトリ

このように，ファイル名，ファイルサイズ，ディスク上の位置などが，すべてディレクトリに記録されているために，ファイルは単なるデータがはいっている部分としてディレクトリの下にしか存在できません．

この階層的ディレクトリ構造を利用することによって，ユーザーは独自の作業環境を作ることができるようになります．たとえば，ハードディスクを複数のユーザーが共用するようなシステムでは，各ユーザーごとのディレクトリ(ホーム・ディレクトリ)を設けることにより，個人別の作業環境を明確にすることができますし，フロッピーディスクをベースにした個人用のシステムであれば，仕事の種類別にディレクトリを分けることによってファイルの整理が行えるため作業効率を上げることもできるのです．

2.3.2　パス名

ファイルが階層的ディレクトリ構造によって管理されているために，ある特定のファイルまたはディレクトリをパス(path)で指定します．これは文字どおり指定するディレクトリやファイルへ到達するための道順を意味しており，この道順を示す文字列をパス名(path name)と呼びます．パス名は〝¥〟※(ルートを意味する)からはじめて，途中のディレクトリを〝¥〟で区切り，最後にディレクトリ名やファイル名をすべて並べて記述します．

Fig. 2.39 におけるいくつかのディレクトリやファイルをパス名で指定する例を次に示しておきます．

```
¥BIN¥CHKDSK . COM
¥TXT¥FILE1
¥COMMAND . COM
```

※ UNIX では，ルートおよびディレクトリやファイルの区切りに〝／〟(スラッシュ)を用いているが，MS-DOS ではマイクロソフト社のスタンダード・バージョンがアスキーコードの〝＼〟(バックスラッシュ：5CH)を割り当てているために，国産マシンの MS-DOS ではパソピア 16 を除いて〝¥〟になっている．

2.3.3　階層的ディレクトリ構造に関するコマンド

MS-DOSには階層的ディレクトリ構造をサポートするために，MKDIR(MaKe DIRectory)，RMDIR(ReMove DIRectory)，CHDIR(CHange DIRectory)の3つのコマンドが用意されていますが，これらはそれぞれ，ディレクトリの作成，削除，移動を行うためのコマンドであり，本項ではFig. 2.39の階層的ディレクトリ構造の作成を例に各コマンドの働きを説明していきます．

この階層構造では，ルート・ディレクトリの下に外部コマンドを集めたBIN，機械語開発ツールを集めたTOOLS，そしてその他のファイルを集めたTXTといったディレクトリがあり，これらのサブ・ディレクトリを作成するとともに，この階層構造の中でファイルのコピー，作成を行います．その際〝COMMAND．COM〟だけは，システムのブート時に不可欠なファイルなので，ルート・ディレクトリに残しておきます．また図では表示されていませんが，〝MSDOS．SYS〟，〝IO．SYS〟もルート・ディレクトリに登録されています．

なお，最初のディスクの状態は，すべてのファイルがルート・ディレクトリにあることを前提としています．

○ディレクトリの作成

MS-DOSをブートした際，ユーザーはルート・ディレクトリに位置しています．ここでMKDIRコマンドを用いてBIN，TOOLS，TXTの3つのディレクトリを作成し，DIRコマンドでその結果を確認しておきます．Fig. 2.40のように，BIN，TOOLS，TXTという名のディレクトリが〈DIR〉という表示で確かめられます．

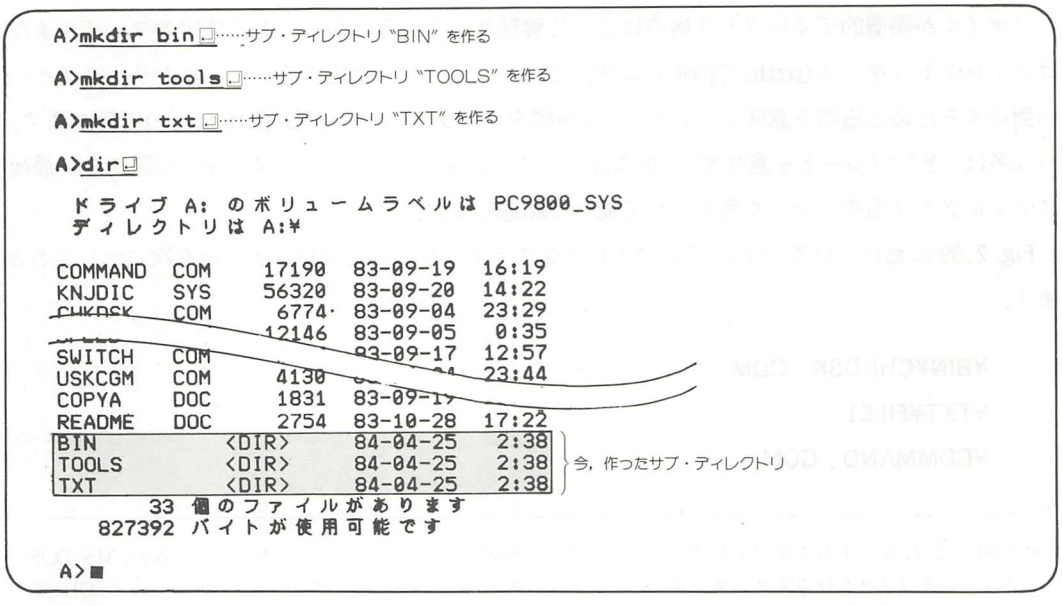


Fig. 2.40　MKDIRコマンドの実行例

次にルート・ディレクトリからサブ・ディレクトリ¥BINに〝CHKDSK．COM〟をコピーしてその結果も確認してみます．この場合，DIRコマンドのパラメータとして¥BINを与えなければなりません(Fig. 2.41)．

```
A>copy chkdsk.com ¥bin↵……カレント・ディレクトリ(ルート)にある〝CHKDSK．COM〟をサブ・ディレクトリ〝BIN〟
        1 個のファイルをコピーしました                          にコピーする

A>dir ¥bin↵……サブ・ディレクトリ〝BIN〟の内容を表示する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥bin

.            <DIR>      84-04-25   2:38 ……カレント・ディレクトリ
..           <DIR>      84-04-25   2:38 ……親ディレクトリ
CHKDSK   COM     6774  83-09-04  23:29 ……コピーされた
        3 個のファイルがあります
   820224 バイトが使用可能です

A>■
```

この2つはMKDIRコマンドでサブ・ディレクトリを作ると，自動的に作成される

Fig. 2.41　パスを指定したCOPYコマンドの実行

図中の〝..〟という表示は，¥BINの親ディレクトリを意味する略記号です．

MS-DOSは，ディレクトリ相互のつながりを管理するために，サブ・ディレクトリを作ると同時に親子のつながりに関するデータを記録したファイルを自動的に作成します．同時に〝.〟は，サブ・ディレクトリ自身を表しますが，これら2つのファイルはシステムが内部的に用いるので，ユーザーはとくに関与する必要はありません．

○ディレクトリの移動

次に，CHDIRコマンドによってディレクトリTOOLSに移り，EDLIN，MASMなどのツールをコピーします．

Fig. 2.42は，EDLINをコピーしたときの状態を表していますが，前回のBINディレクトリへファイルをコピーした場合と異なり，コピー先のファイル名にではなくコピーもとのファイル名にパスの指定があることに注目してください．

階層的ディレクトリ構造の中でユーザーが作業を行う場合CHDIRコマンドによってディレクトリを移動していくために，ユーザーが位置するディレクトリ(カレント・ディレクトリ)が，つねにパス名の指定やコマンドの働きの基準となっているのです．この場合は，カレント・ディレクトリがTOOLSになっているために，コピー先のファイル名のパス名が省略可能になっています．

```
A>chdir tools ↵……サブ・ディレクトリ "TOOLS" にカレント・ディレクトリを変更する

A>copy ¥edlin.com edlin.com ↵……ルート・ディレクトリにある "EDLIN.COM" をカレント・ディレクトリ "TOOLS"
        1 個のファイルをコピーしました                                    にコピーする

A>dir /w ↵……/Wスイッチを付けて，ディレクトリの内容を表示する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥tools

.          ..          EDLIN     COM ……コピーされた
        3 個のファイルがあります
   811008 バイトが使用可能です

A>■
```

Fig. 2.42　カレント・ディレクトリとパスの指定

○パスの相対指定と絶対指定

Fig. 2.42 の実行例が終了した状態では，カレント・ディレクトリは TOOLS になっていますが，ここで TXT にカレント・ディレクトリを移してみます．このとき，CHDIR に続くパス名の指定には次に示すような 2 通りの指定方法を用いることができます．

CHDIR　. . ¥TXT……相対指定
CHDIR　¥TXT…… 絶対指定

相対指定とは，カレント・ディレクトリを基準にしてパス名の指定を行うものであり，上の例では〝..〟で親ディレクトリ(ルート・ディレクトリ)へ移り，その後で TXT へ移っています．これに対し絶対指定では，カレント・ディレクトリの位置とは無関係に，つねにルート・ディレクトリからのパス名を指定するものです．

ところで，頻繁にディレクトリを移動していると，現在位置しているディレクトリがわからなくなってしまうことがあります．このような場合，カレント・ディレクトリの位置を確認するためには，次のように単に CHDIR コマンドだけを入力します．

A > CHDIR ↵

Fig. 2.43 では，カレント・ディレクトリを TXT に移してディレクトリの位置を確認しています．

```
A>chdir ..¥txt ↵……カレント・ディレクトリを親ディレクトリにある "TXT" に変更する．この場合，親ディレクトリがルートなので
                   CHDIR ¥TXTでもよい
A>chdir ↵……カレント・ディレクトリを表示する
A:¥txt ……カレント・ディレクトリは¥TXT

A>■
```

Fig. 2.43　パスの確認

○コマンドとパス

それでは，カレント・ディレクトリ¥TXT の下にファイルを作成してみることにします．

まず，ファイル名を〝FILE1〟として EDLIN を起動させます(Fig. 2.44)．

```
A>edlin file1 ……EDLINを起動してファイル〝FILE１〟を作成する
コマンドまたはファイル名が違います ……EDLINがカレント・ディレクトリにないのでエラーになった

A>■
```

Fig. 2．44　EDLIN起動エラー

残念ながらエラーになってしまいます．これは，コマンド・プロセッサのコマンドを呼び出す(検索する)パスが限定されているために発生しているものです．つまりコマンド・プロセッサは，外部コマンドが入力されるとまずカレント・ディレクトリで目的のファイルを捜すようになっているために，そのファイルが未登録の場合にはエラーとなってしまうわけです．

この場合，EDLIN は¥TOOLS に登録されているので，このままではこのディレクトリにおいてのみ EDLIN が使用できることになりますが，これでは階層的ディレクトリ構造の意味がまったくありません．この状態を解消するためには，コマンド・プロセッサに対してカレント・ディレクトリのほかに¥TOOLS も検索するように指示する必要があります．このようなコマンドを検索する経路を〝サーチパス〟と呼び，これは PATH コマンドで設定します．

コマンド・プロセッサが¥TOOLS に登録されているファイルを対象とするためには，次のように入力します．

```
PATH　¥TOOLS ↵
```

これで，カレント・ディレクトリの位置にかかわらず，コマンド・プロセッサは¥TOOLS を検索するようになります．さらに，¥BIN に登録されている外部コマンドも，コマンドのサーチパスの対象とするためには，ディレクトリ名をセミコロン(；)で区切って複数指定します．

```
PATH　¥TOOLS ; ¥BIN ↵
```

いったん設定したサーチパスは，再度指定するかあるいは電源を切るまで有効に働きます．サーチパスの設定状況を確認するのであれば，PATH コマンドのみを入力します．Fig. 2.45 は，現在のコマンドのサーチパスを表示したものです．

```
A>path ¥tools;¥bin ……パスを〝¥TOOLS〟と〝¥BIN〟に設定する．これ以降，外部コマンドがカレント・ディレクトリに存在
                      しない場合〝¥TOOLS〟，〝¥BIN〟の順で外部コマンドが検索される
A>path ……現在設定されているパスを表示する
PATH=¥TOOLS;¥BIN ……パスが設定された

A>■
```

Fig. 2．45　サーチパスの確認

さて以上の設定によって，MS-DOSのすべての外部コマンドが，カレント・ディレクトリの位置にかかわらず利用できるようになりました．再度EDLINを起動したものがFig. 2.46ですが，今度は正常に働いています．

```
A>edlin file1 ↵……EDLINでファイル "FILE1" を作成する
新しいファイルです……今度はEDLINが起動した
*i ↵……インサートコマンド
     1:*Now, EDLIN comes up !! ↵
     2:*^Z ↵……CTRL-Zを入力してインサートモードを抜ける
*e ↵……ファイルをセーブしてEDLINを終了する

A>■
```

Fig. 2.46　正常なEDLINの起動

○ディレクトリの削除

仕事の内容に応じてサブ・ディレクトリを作成していくと，ファイル管理が簡単になります．仕事が終わってそれらのファイルを当分用いなくなったら，必要なファイルのバックアップをとって，サブ・ディレクトリごと削除してしまいます．

ディレクトリを削除するには，RMDIRコマンドを用います．ただし，このコマンドが働くには，次の2つの条件を満たしている必要があります．

(1)削除するディレクトリに，ユーザーファイルやサブ・ディレクトリが登録されていないこと．すなわちディレクトリを調べたとき，"." と ".." だけになっていなければならない

(2)カレント・ディレクトリは削除できない．したがって，離れたディレクトリ——たいてい親ディレクトリ——からコマンドを入力する

したがってディレクトリの削除は，まずすべてのファイルの削除を行い，次にディレクトリを削除するという2ステップの処理になります．

ディレクトリ¥TXTを削除した例をFig. 2.47に示します．

```
A>del txt ↵……サブ・ディレクトリ "TXT" にあるすべてのファイルを削除する
よろしいですか <Y/N>? y ↵……DELコマンドのパラメータに "*.*" またはパス名を指定すると，そのディレクトリにあるすべ
                            てのファイルが削除の対象となり，メッセージが表示されて確認が求められる
A>rmdir txt ↵……サブ・ディレクトリ "TXT" を削除する

A>dir t* ↵……"T" ではじまるファイル，ディレクトリを表示する

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

TOOLS          <DIR>          84-04-25   2:38  ……サブ・ディレクトリ "TXT" は削除されている
       1 個のファイルがあります
  812032 バイトが使用可能です

A>■
```

Fig. 2.47　RMDIRコマンドの実行例

すべてのファイルを削除する場合には，DEL コマンドに〝*．*〟を与えることによって処理できることは，前にも述べたとおりですが，上の例では DEL コマンドにディレクトリ名を指定することで，その中のすべてのファイルを削除しています．前のコマンドと同様，必要なファイルまで削除してしまう事故を防止するために確認を求めるメッセージが表示されます．

2.4 デバイスファイル

2.2 でも述べたように，MS-DOS ではデータの入出力の対象となるものをすべてファイルとして扱っています．つまり，デバイスとディスク上に作成されるファイルとがまったく同等に扱われるのです．

MS-DOS では，ディスク以外に標準でサポートされるデバイスとしてスクリーン，キーボード，補助入出力，プリンタがありますが，これらのデバイスはすべて Table 2.5 に示すようなデバイス名が割り当てられています．

Table 2．5　デバイスファイル

デバイス名	デバイス	機　能
CON	キーボードとスクリーン	CONはコンソール(CONsole)を意味している．このデバイス名を指定した場合は，入力はキーボードから出力はスクリーンが割り当てられる
PRN	プリンタ	プリンタ出力用のデバイス名．転送もと（データの送り側）にPRNを指定するとエラーとなる
AUX	補助入出力	AUX(AUXiliary)はRS-232C などの補助入出力を指定する
NUL	ダミー出力	ダミー出力用デバイス名．このデバイス名を出力に指定すると，出力の内容をNULがすべて吸収してファイルの内容はどこにも残らない．入力に指定した場合はEOFが返る

2.4.1　デバイスファイルの使い方

2.2 では，COPY コマンドの機能のうちファイルのコピーと結合について説明しましたが，ここではもう１つの機能であるファイルの転送機能を用いてデバイスファイルの使用例を示します．

○コンソール(CON)

Fig. 2.48 の例では転送もとのファイル名に CON を指定しているために，キーボードから入力したテキストをそのまま転送先のファイルの内容とします．テキストはリターンキーの入力(0DH)によって区切られ，CTRL-Z(1AH)の入力によって終了してコマンドレベルに戻ります．〝NEWFILE〟がディスクに登録されていないファイルであれば新たにファイルが作成されますが，すでに登録されているファイルである場合にはファイルの内容が更新されてしまいます．なお 2.2 ではファイルの作成に EDLIN を使用していますが，小さなテキストファイルであれば

COPY コマンドによって手軽に作成することができます.

```
A>copy con newfile↵……キーボードからの入力をファイル "NEWFILE" にコピーする
You can create files easily by means of 'copy con <filename>'.↵……キーボードからテキストを入力する
^Z↵……CTRL-Zを入力して，DOSのコマンドレベルに戻る
        1 個のファイルをコピーしました

A>■
```

Fig. 2.48　CONデバイスの指定

Fig. 2.49 で示した例は，Fig. 2.48 で示した逆，つまり出力に CON を指定したものですが，実行例を見ればわかるように TYPE コマンドと同様の結果が得られます.

```
A>copy newfile con↵
You can create files easily by means of 'copy con <filename>'.……スクリーンに表示される
        1 個のファイルをコピーしました

A>■
```

Fig. 2.49　出力にCONを指定した場合

○プリンタ(PRN)

PRN を COPY コマンドの転送先のファイル名に指定することによって，コンソールやファイルからの入力をそのままプリンタに出力することができます．プリンタ出力用のコマンドとしては PRINT コマンドがありますが，PRINT コマンドが改ページを行うのに対し，Fig. 2.50 の実行例は改ページを行いません.

```
A>copy newfile prn↵……今作ったファイル "NEWFILE" をプリンタに出力する
        1 個のファイルをコピーしました

A>■
```

```
You can create files easily by means of 'copy con <filename>'.
                                    ……プリンタに出力されたファイル "NEWFILE" の内容
```

Fig. 2.50　PRNデバイスの指定

○補助入出力(AUX)

AUX は，補助入出力用デバイスとして通常 RS-232C が割り当てられています．MS-DOS では標準としてサポートしているデバイスファイルですが，RS-232C がオプションとなっている機種では利用することはできません.

Fig. 2.51 では，ファイルを AUX に出力していますが，補助出力に用いる場合にはデータの出

力側および受け側の双方にRS-232Cの初期設定が必要なのはいうまでもありません.

なお，COPYコマンドでは次のようにAUXを転送元に指定しても動作しませんが，メーカーによっては補助入力用のCOPYコマンドをサポートしているものもあります(PC-9800シリーズのCOPYAコマンド).

```
COPY  AUX  FILE
```

```
A>copy newfile aux ↵……ファイル "NEWFILE" を補助出力装置(AUX)に転送する
        1 個のファイルをコピーしました

A>■
```

Fig. 2.51　AUXデバイスの指定

○ダミー出力(NUL)

NULはダミー出力用のデバイスファイルであるため，コマンドレベルで用いることはあまり考えられませんが，出力の制御を行うプログラムを作る場合には利用されています．たとえば身近なところでは，MASMの出力制御に用いられていることが挙げられるでしょう.

2.4.2　標準入出力とI／Oのリダイレクト(入出力の切換え)

これまで述べてきたように，入出力の対象がすべてファイルとして同格に扱われているとすれば，DOSがデータの出入口がどこであるかを管理しておきさえすれば，プログラム(コマンド)では入出力の先がどこであるかを意識する必要がなくなります．MS-DOSでは，Fig. 2.52のように入出力を標準入出力として一本化して扱い，標準入出力に対する方向(データの出入口)をリダイレクト(切換え)することによって入出力先のファイルを容易に変更することができます.

標準入出力は，デフォルトではキーボードとスクリーンに割り当てられていますが，プログラム(コマンド)側で標準入出力を利用している限り，ユーザーは自由にI／Oのリダイレクトを行うことができます.

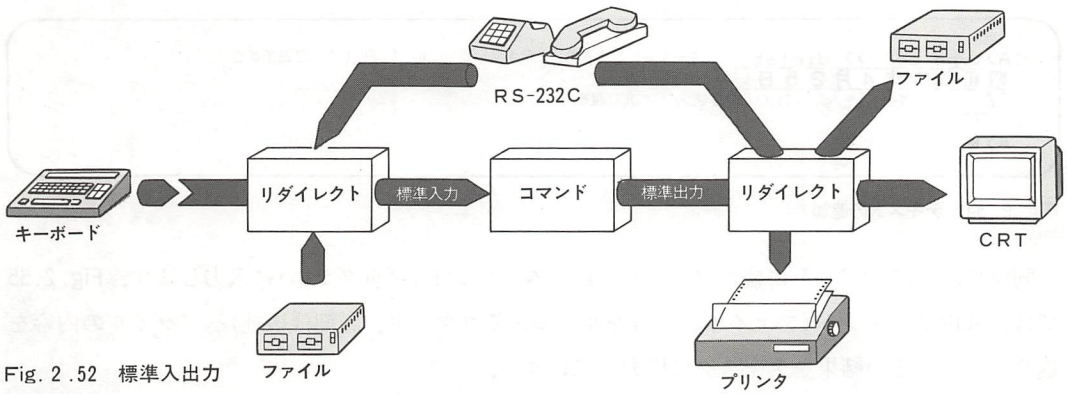


Fig. 2.52　標準入出力

○出力のリダイレクト

それでは，いままでに登場したコマンドを例に，実際にI／Oのリダイレクトを行って結果を確認してみることにします．

たとえば，DIRコマンドはファイル名の一覧をスクリーンに対して出力するものですが，Fig. 2.53のように出力を切り換えることによってディスクに出力することができます．

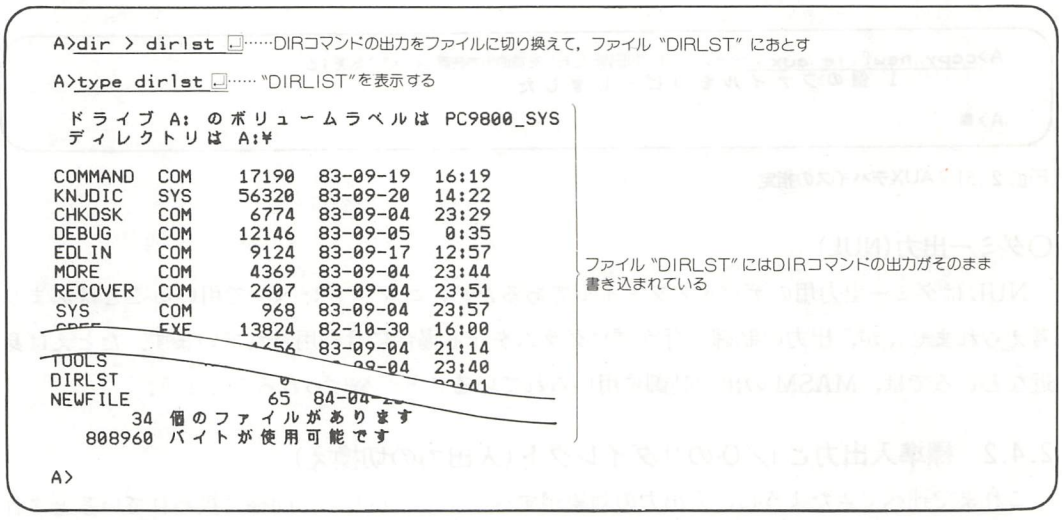


Fig. 2.53　出力のリダイレクト

"＞" 以下の示す意味はスクリーンではなく "＞" に続くファイル(この場合はDIRLST)，あるいはデバイスファイルに出力せよという意味になります．その際，指定したファイルが存在しなければ新しいファイルが作成されますが，すでに存在している場合はもとのファイルの内容が更新されてしまいますので，もしすでに存在するファイルの終わりに新しい内容を追加するのであれば，"＞＞"を使用しなければなりません．Fig. 2.54では，TYPEコマンドによってコンソールからファイル "DIRLST" にテキストを追加しています．

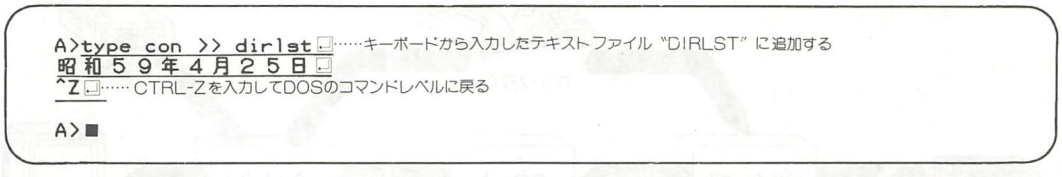


Fig. 2.54　テキストの追加

同様に，"＜" は "＜" に続くファイルの内容をコマンド(プログラム)に入力します．Fig. 2.55では，SORTコマンド(ファイルの内容を並べ換えるコマンド．詳細は後述)へファイルの内容を送り，さらにその結果をディスクに出力しています．

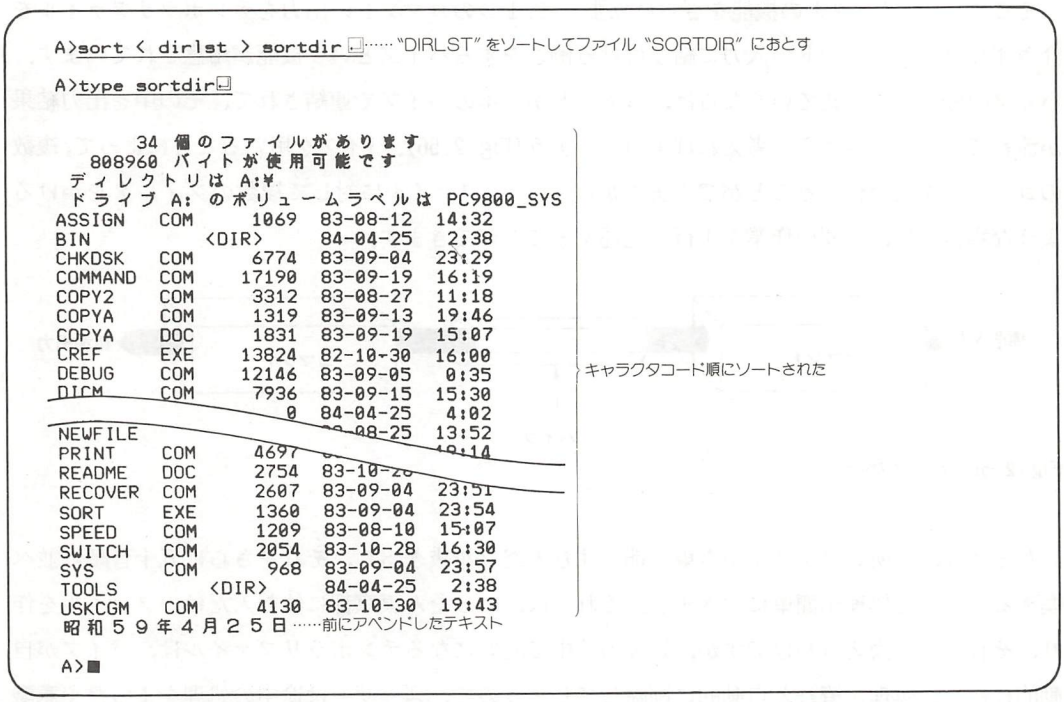


Fig. 2.55　入力のリダイレクト

2.5 フィルタとパイプ

フィルタとは，標準入力からデータを読み取り，それに対して単純な変換を行って結果を出力する，一種の「ふるい」の役割を果たすコマンドのことをいいます．

MS-DOS には次の 3 種類のフィルタが用意されていますが，UNIX などでは，このほかにもスペルチェックやワードカウントといったフィルタを数多く備えています．

FIND　……検索
MORE　……1 画面ごとのスクリーン出力
SORT　……並換え

各種フィルタは，標準入力からデータを読み込み，標準出力へ出力するようになっていますから，I／O のリダイレクトによってファイルの入出力を制御することができます．しかし，前節の I／O のリダイレクトで見てきたように，2 つ以上のコマンドを使って出力結果を得る場合，コマンドの出力をいったんテンポラリファイルに出力して，別のコマンドに入力するという形をとらなければなりません．テンポラリファイルは，ほかに何の使い道があるわけでもなく，このようなファイルを作るのは単に処理がわずらわしくなるだけです．

そこで，リダイレクトの機能をさらに拡張して，1つのコマンドの出力をテンポラリファイルを介さずにほかのコマンドの入力に結び付ける働きをするパイプという機能が用意されています．パイプの働きをたとえていうならば，コマンドが1本のパイプで連結されて，その中を出力結果が流れていくというように考えればよいでしょう(Fig. 2.56)．パイプを用いることによって，複数のコマンドを結び付けることができますから，1つのファイルに対して複数のフィルタをかけるような場合には，一連の作業を1行で記述することができます．

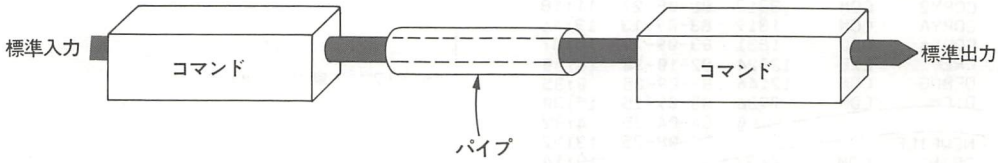


Fig. 2.56　パイプの処理

たとえば，住所録ファイルから東京都に住む人だけを抜き出し，それをさらに五十音順に並べ換えるといった処理が簡単にできます．それには，いったん東京都に住む人だけのファイルを作り，それを並べ換えるわけですが，処理の途中で必要になるテンポラリファイルは，パイプが自動的に作り，処理が済むと自動的に削除してしまうので，ユーザーは途中の処理をまったく意識することがなくフィルタの機能だけを利用することができます．

それでは，Fig. 2.57 のような住所録のファイルを例に，フィルタを使っていろいろ操作してみましょう．

```
A>type address↵……ファイル "ADDRESS" の内容を表示する
東京都　三鷹市　牟礼 XXXXXXXXXXXXXXXXXXXXXXXX
東京都　港区　南青山 XXXXXXXXXXXXXXXXXXXXXXX
東京都　武蔵野市　吉祥寺 XXXXXXXXXXXXXXXXXXXX
東京都　杉並区　久我山 XXXXXXXXXXXXXXXXXXX
茨城県　取手市 XXXXXXXXXXXXXXXXXXX
千葉県　我孫子市 XXXXXXXXXXXXXXXXXXXXX
埼玉県　浦和市 XXXXXXXXXXXXXXXXX
神奈川県　横浜市 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
神奈川県　川崎市 XXXXXXXXXXXXXXXXXXXXXXXXX
東京都　荒川区　東日暮里 XXXXXXXXXXXXXXXXXXX
東京都　文京区　小日向 XXXXXXXXXXXXXXXXXXXXX

A>■
```

Fig. 2.57　ADDRESSファイルの内容

○ファイルの並換え(SORT)

コンピュータで扱うことのできる文字には，すべてコード番号が付けられていますが，MS-DOSではASCIIコード，マイクロソフト漢字コードの2種類のコード体系を採用しています．

SORTコマンドは，ファイルの各ラインをこの文字コード順に並べ換える働きをするもので

す．ですから，SORT コマンド実行後は，ファイルの内容はちょうど辞書のように並べ換えられて出力されます．

ここで注意しなければならないのは，漢字の並換えです．漢字は，一種類の読み――たいていは音読み――だけにコード番号が付けられているので，かならずしも意図した読みどおりには並べ換えられません．

では，住所録ファイル〝ADDRESS〟を SORT コマンドに通してみましょう(Fig. 2.58)．

```
A>type address ¦ sort□……パイプ処理を用いて，ファイル〝ADDRESS〟をソートする
茨城県　取手市 XXXXXXXXXXXXXXXXXXX
埼玉県　浦和市 XXXXXXXXXXXXXXXXXX
神奈川県　横浜市 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
神奈川県　川崎市 XXXXXXXXXXXXXXXXXXXXXXXX
千葉県　我孫子市 XXXXXXXXXXXXXXXXXXXX
東京都　港区　南青山 XXXXXXXXXXXXXXXXXXXXXX
東京都　荒川区　東日暮里 XXXXXXXXXXXXXXXXXXX
東京都　三鷹市　牟礼 XXXXXXXXXXXXXXXXXXXXXX
東京都　杉並区　久我山 XXXXXXXXXXXXXXXXXXX
東京都　武蔵野市　吉祥寺 XXXXXXXXXXXXXXXXXXX
東京都　文京区　小日向 XXXXXXXXXXXXXXXXXXXX

A>■
```

Fig. 2.58　ADRESS ファイルのソート

TYPE コマンドの出力には標準出力が用いられているために，通常はスクリーンに表示されますが，パイプでこの出力を SORT コマンドの入力に結び付けると，SORT コマンドの出力も標準出力ですから，並べ換えた結果はやはりスクリーンに表示されます．

この例では，ADDRESS ファイルの内容を TYPE コマンドでスクリーンに出力しましたが，次のように I／O のリダイレクトを用いても同様の結果が得られます．

```
SORT < ADDRESS
```

これで，ADDRESS ファイルの内容が I／O のリダイレクトによって，直接 SORT コマンドに送られます．

SORT コマンドは標準出力に出力するコマンドですから，I／O のリダイレクトで Fig. 2.59 のようにディスクファイルに切り換えると，スクリーンと同じイメージのファイル〝ADDSORT〟が得られます．

また，次のコマンドもまったく同じ働きをします．

```
SORT < ADDRESS > ADDSORT
```

SORT コマンドの入力は，標準入力であれば何でもかまいません．たとえば DIR コマンドの出力結果も標準出力へ送られるため，次のように入力すると DIR コマンドの出力がパイプで

```
A>type address ¦ sort > addsort ↵……パイプを用いて〝ADDRESS〟をソートし，その結果をファイル〝ADDSORT〟に出力する
A>type addsort ↵……ファイル〝ADDSORT〟を表示する
茨城県　取手市　XXXXXXXXXXXXXXXXXXX
埼玉県　浦和市　XXXXXXXXXXXXXXXXXX
神奈川県　横浜市　XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
神奈川県　川崎市　XXXXXXXXXXXXXXXXXXXXXXXXXX
千葉県　我孫子市　XXXXXXXXXXXXXXXXXXXXX
東京都　港区　南青山　XXXXXXXXXXXXXXXXXXXXXX
東京都　荒川区　東日暮里　XXXXXXXXXXXXXXXXXXX
東京都　三鷹市　牟礼　XXXXXXXXXXXXXXXXXXXXXX
東京都　杉並区　久我山　XXXXXXXXXXXXXXXXXXX
東京都　武蔵野市　吉祥寺　XXXXXXXXXXXXXXXXXX
東京都　文京区　小日向　XXXXXXXXXXXXXXXXXXXX

A>■
```

Fig. 2.59　パイプとリダイレクトの組合せ

SORT コマンドに送られ，ファイル名がコード順に並べ換えられて表示されます．

```
DIR | SORT
```

このとき，〝SORT＜DIR〟という使い方はできません．というのは〝＜〟〝＞〟の後ろはかならずファイルでなければならないからです．

SORT コマンドの／＋n スイッチを用いると，並換えの対象となる文字の位置を〝n 文字目以降〟と指定することができます．前例ではこのスイッチを省略したために各ラインの先頭の文字が並換えの対象となり，その結果ファイル名がアルファベット順に並べ換えられた出力が得られました．ディレクトリ表示の 14 カラム(文字)目以降について並換えを行うように

```
DIR | SORT　／＋ 14
```

と入力すると，Fig. 2.60 のようにファイルの大きさ順の出力が得られます．

SORT コマンドには，もう 1 つ／R スイッチがありますが，これはコード番号の逆順に並換えを行います．

```
A>dir ¦ sort /+14 ↵……パイプ処理を用いて，ディレクトリの内容を先頭から14文字以降の文字(ファイルの大きさ)でソートする．ファイルの大きさが小さい順にソートされる
                  →これ以降の文字でソートする
%PIPE1   $$$          0   84-04-25    4:44 }
%PIPE2   $$$          0   84-04-25    4:44 }……パイプ処理のためのテンポラリファイル
NEWFILE              65   84-04-25    3:47
ADDSORT             479   84-04-25    4:41
ADDRESS             480   84-04-25    4:38
SYS      COM        968   83-09-04   23:57
ASSIGN   COM       1069   83-08-12   14:32
SPEED    COM       1209   83-08-10   15:07
COPYA    COM       1319   83-09-13   19:46
SORT     EXE       1360   83-09-04   23:54
DIRLST              1520   84-04-25    4:07
                    1520   84-04-25    4:29
                           83-09-04   21:14
```

```
CREF       EXE     13824
COMMAND    COM     17190   83-07-21
LIB        EXE     32128   82-10-30   16:00
LINK       EXE     42624   83-08-22   15:40
KNJDIC     SYS     56320   83-09-20   14:22
MASM       EXE     80256   83-08-12   14:18
BIN              <DIR>      84-04-25    2:38
TOOLS            <DIR>      84-04-25    2:38
   802816 バイトが使用可能です
 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥
      39 個のファイルがあります

A>■
```

Fig. 2.60　SORTコマンドのカラム指定

○文字列の検索(FIND)

FINDコマンドには，標準入力からはいるファイル中の指定した文字列を含む行だけを通す働きがありますが，この機能を用いると，簡単に検索を行うことができます．例として，〝ADDRESS〟ファイルの中から東京都だけを抜き出して表示したものをFig. 2.61に示します．なお，検索する文字列は引用符(〝　〟)で囲みます．

```
A>type address ¦ find "東京都"↵……パイプ処理を用いて，ファイル〝ADDRESS〟の中の〝東京都〟を含む行を検索する
東京都　三鷹市　牟礼 XXXXXXXXXXXXXXXXXXXXXXX
東京都　港区　南青山 XXXXXXXXXXXXXXXXXXXXXXX
東京都　武蔵野市　吉祥寺 XXXXXXXXXXXXXXXXXXXX
東京都　杉並区　久我山 XXXXXXXXXXXXXXXXXXX
東京都　荒川区　東日暮里 XXXXXXXXXXXXXXXXXXX
東京都　文京区　小日向 XXXXXXXXXXXXXXXXXXXXX

A>■
```

Fig. 2.61　FINDコマンドの実行例

また，I／Oのリダイレクトによって次のようにしても，同様の出力を得ることができます．

```
FIND 〝東京都〟 <　ADDRESS
```

FINDコマンドには，複数のファイルをオープンして読み込む機能が備えられており，この機能を用いるとI／Oのリダイレクトを行わずに，直接フィルタを通すことができます．

それには，次のようにしてドライブ〝A：〟，〝B：〟のディレクトリを収めたファイルを作っておき，2つのファイルからFINDコマンドによってファイル名に〝．EXE〟の付いたファイルを検索します(Fig. 2.62)．

そのほか，SORTコマンドと同様にI／Oのリダイレクトを用いれば，Fig. 2.62のスクリーン出力がそのままのイメージでファイル〝DIRAB〟に収められます．

FINDコマンドにはこのほかにも，／V，／C，／Nの3つのスイッチがありますが，詳細は第II部を参照してください．

```
A>dir a: > dira ↵……ドライブAのカレント・ディレクトリの内容をファイル "DIRA" におとす

A>dir b: > dirb ↵……ドライブBのカレント・ディレクトリの内容をファイル "DIRB" におとす

A>find "EXE" dira dirb > dirab ↵……ファイル "DIRA", "DIRB" のうち "EXE" を含む行を検索し，ファイル
                                    "DIRAB" におとす
A>type dirab ↵……"DIRAB" をタイプする

---------- dira
CREF     EXE    13824  82-10-30  16:00
EXE2BIN  EXE     1656  83-09-04  21:14
FC       EXE     2642  83-09-04  23:40
FIND     EXE     6430  83-09-04  23:41   } "DIRA" のうち "EXE" を含む行
LIB      EXE    32128  82-10-30  16:00
LINK     EXE    42624  83-08-22  15:40
MASM     EXE    80256  83-08-12  14:18
SORT     EXE     1360  83-09-04  23:54

---------- dirb
FACT     EXE    13496  84-03-05  19:14
FIBBO    EXE    13496  84-02-22   5:45
ACKERF   EXE    13530  84-02-25  12:09   } "DIRB" のうち "EXE" を含む行
TOWER    EXE    14504  84-03-05  17:19
LFACT    EXE    13712  84-03-08  17:19

A>■
```

Fig. 2.62　FINDコマンドの複数ファイルの読込み

○1画面ごとのスクリーン出力(MORE)

MOREコマンドは，標準入力からはいるファイルの内容をスクリーンの1画面分――23行に区切りながら標準出力へ出力する働きがあります．何画面分ものファイルをCTRL-Sで停止させながらTYPEコマンドで読むのは実用的ではありませんので，通常はこのMOREコマンドに送って表示させます．

例として，〝ADDRESS〟ファイルをMOREコマンドによって出力した例をFig. 2.63に示します．

MOREコマンドは，CRコード(0DH)を23個通したところでいったん出力を停止します．その際出力すべきものがまだ残っていると，スクリーンの最下行には

```
--MORE--
```

と表示され，CTRL-C以外の任意のキー(スペースキーなど)を押すと次の23行が表示されます．

なお，次のようにパイプを用いても同様の結果が得られますが，このコマンドの実行ではいったんテンポラリファイルが作成されるために，リダイレクトを用いるよりも時間がかかります．

```
TYPE  ADDRESS | MORE
```

このMOREコマンドは，UNIXから取り入れられた機能ですが，実はUNIXより若干機能が縮小されており，〝MORE 〈ファイル名〉〟といった使い方ができなくなっているほか，CRコー

ドを数えているだけなので1行が80文字(スクリーンの横の文字数)を超えている場合，不都合が生じてしまいます．

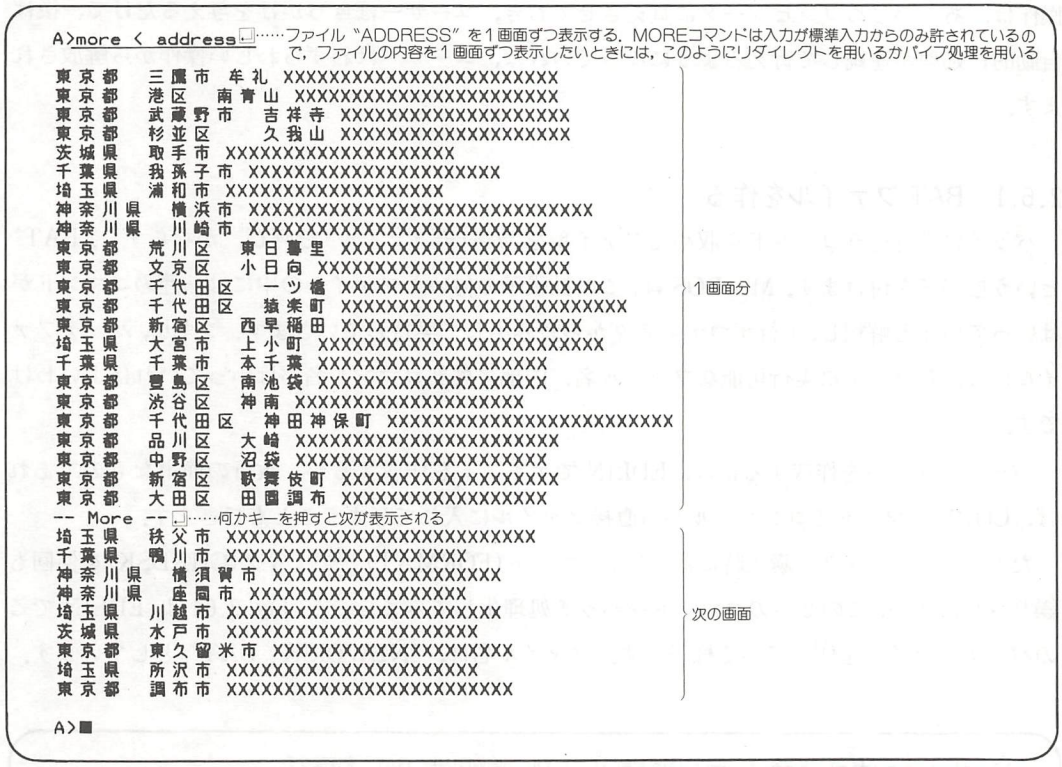


Fig. 2．63　MOREコマンドの実行例

2.6 バッチ処理

バッチ処理とは，あらかじめコンピュータに一連の操作を指定しておき，何回も繰り返す決まりきった処理は，それに沿って自動的に実行するというものですが，DOSのバッチ処理機能は，ユーザーにどのようなメリットをもたらすのでしょうか．たとえば，プログラムを開発するとき，ソースファイルの作成→コンパイル→実行テストという手順を何回も繰り返します．コンパイルや実行中にエラーが発生すると，原因を解明し，ソースファイルを修正して，再びコンパイルという作業を何回も繰り返しながらプログラムは完成していきます．このとき，プログラマーが必要とするのは，ソースファイルのリストと，コンパイル中のエラーメッセージ，実行結果くらいです．

ソースファイルの入力と修正は，ユーザーの手をわずらわせるしかありませんが，そのほかにもコンピュータに対する指示がいくつか必要です．コンパイラに対するソースファイルの指定，

リンカに対する指示，エラーメッセージのプリントアウト，動作確認といったプログラムの内容とは直接関係のない操作を何回も繰り返すのは，何ともめんどうな仕事です．

このようなよごれ仕事は，コンピュータにできるだけ負担させるに限ります．何回も繰り返す操作は，あらかじめコンピュータに覚えさせておき，ユーザーはきっかけを与えるだけで，後は自動的に処理が連続して行えるようになっていれば，ユーザーはわずらわしい操作から解放されます．

2.6.1　BATファイルを作る

バッチ処理を行うコマンドを収めたファイルを〝バッチファイル〟と呼び，かならず〝．BAT〟という拡張子を付けます．MS-DOSは，この拡張子の付いたファイルの中には一連のコマンドがはいっていると解釈し，1行ずつディスクから読み出して実行していきます．つまり，バッチファイルには，各ラインに実行可能なファイル名，内部／外部コマンド名がはいっていればよいわけです．

バッチファイルを作成するには，EDLINでテキストを作成するか，数行の簡単なものであれば，COPYコマンドでコンソールから直接ファイルに入れてしまうこともできます．

たとえば，ディスクの購入時には，フォーマット(FORMAT)とチェック(CHKDSK)を何回も繰り返しますが，この2つのコマンドをバッチ処理化してみましょう．Fig. 2.64は，EDLINでこのバッチファイルを作っている様子です．ファイル名は，〝NEWDISK．BAT〟としています．

```
A>edlin newdisk.bat↵……EDLINを起動してファイル〝NEWDISK．BAT〟を作成する
新しいファイルです
*i↵……インサート・コマンド
     1:*format b: /v/s↵……{ドライブBのディスクをフォーマット／V／Sスイッチはそれぞれボリュームラベルを付ける，
     2:*chkdsk b:↵        システムを転送する
     3:*^Z↵
*e↵……ファイルをセーブしてEDLINを終了する

A>dir newdisk.bat↵

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

NEWDISK  BAT        28  84-04-25   6:50  ……今作成したファイル
       1 個のファイルがあります
  800768 バイトが使用可能です

A>■
```

Fig. 2.64　バッチファイルの作成

これで，FORMATとCHKDSKコマンドが〝NEWDISK．BAT〟ファイルにはいりました．このバッチファイルを実行するには，コマンドを入力するときと同じように，次のようにファイル名を入力します．

A > NEWDISK ↵

拡張子〝．BAT〟は，省略することができます．これだけの入力で，2つのコマンドをいちいちキーボードから入力したときと同じ処理が行えます．

Fig. 2.65の例は，マイクロソフト社の販売しているFORTRAN(MS-FORTRAN)で，TESTというプログラムを開発するときのものです．

```
A>type devtest.bat ↵……ファイル〝DEVTEST．BAT〟を表示する
edlin test.for……EDLINを起動して〝TEST．FOR〟を作成する
more < test.for ……MOREコマンドで〝TEST．FOR〟の内容を確認する
b:for1 test.for;……FORTRANコンパイラpass one
b:pas2 ……FORTRANコンパイラpass two
b:link test,test,nul,b:fortran……リンカで実行可能なファイル〝TEST．EXE〟を生成する
test ……今作成した〝TEST．EXE〟を実行する

A>■
```

Fig. 2．65　FORTRANプログラム用バッチファイル

1行目でEDLINを起動し，〝TEST．FOR〟というソースファイルを作ります．いったんEDLINが起動したら，後は，まったく普通にEDLINを使い，ファイルが完成したところでEndコマンドを入力すると，このファイルがディスクにセーブされます．MS-DOSは，続いて2行目以降の処理に移ります．2行目では，MOREコマンドで，〝TEST．FOR〟の内容を確認しています．3〜5行目のコマンドは，最もオーソドックスなMS-FORTRANによるコンパイルの手順です．FOR1，PAS2のコンパイルとリンク処理が済むと，〝TEST．EXE〟という実行可能なプログラムファイルが完成します．6行目でこれを実際に動かして結果を確かめています．ソース・プログラムの入力は，ユーザーの手をわずらわせない限り不可能ですが，それ以外のコマンド入力からは，最初に〝DEVTEST〟と入力するだけで，いっさい解放されます．

また，バッチ処理を途中で中止したいときがあります．たとえば，前例の2行目で，ソースファイルを確認している最中に誤りに気付いたときなどです．このようなときは，処理の途中でCTRL-Cを入力します．するとFig. 2.66に示すメッセージによってバッチ処理の中止の確認を求められます．〝Y〟を選択すると，処理が打ち切られ，MS-DOSのコマンドレベルに戻ります．〝N〟を選択すると，MS-DOSは次のラインから処理を再開します．

また，一度中断した処理を再開するときは，次のラインから処理が再開される点に注意してください．前例の3行目，コンパイルの第1段階(FOR1 TEST)の途中でCTRL-Cで中断し，再度実行を開始すると，この処理が完了していないまま第2段階からバッチ処理は再開しますから，当然正しいコンパイル結果は得られなくなってしまいます．

```
A>devtest↵……バッチファイル "DEVTEST" を実行する

A>edlin test.for↵……自動的にバッチファイルから出力される
ファイルを終りまで読込みました
*i↵……インサートコマンド
     1:*        integer ipos(8),ivrt(8),rcrs(15),lcrs(15)↵
     2:*        do 10 i=1,8↵
     3:*          ivrt(i)=0↵
     4:*   10   continue↵
     5:*        do 20 i=1,15↵
     6:*          rcrs(i)=0↵
     7:*          lcrs(i)=0↵
     8:*   20   continue↵
     9:*        i=1↵
    10:*   30   n=0↵
    11:*   40   n=n+1↵
    12:*        if (n .gt. 8) goto 60↵
    13:*        if (ivrt(n) .eq. 1) goto 40↵
    14:*        if (rcrs(i-n+8) .eq. 1) goto 40↵
    15:*        if (lcrs(i+n-1) .eq. 1) goto 40↵
    16:*          ipos(i)=n↵
    17:*          ivrt(n)=1↵
    18:*          rcrs(i-n+8)=1↵
    19:*          lcrs(i+n-1)=1↵
    20:*          i=i+1↵
    21:*        if (i .le. 8) goto 30↵
    22:*        write(*,50) (ipos(iy),iy=1,8)↵
    23:*   50     format(' answer=',8i3)↵
    24:*   60   i=i-1↵
    25:*        n=ipos(i)↵
    26:*        ivrt(n)=0↵
    27:*        rcrs(i-n+8)=0↵
    28:*        lcrs(i+n-1)=0↵
    29:*        if (i .ge. 1) goto 40↵
    30:*        end↵
    31:*^Z↵
*e↵……EDLINを終了する

A>more < test.for……自動的にバッチファイルから出力される

      integer ipos(8),ivrt(8),rcrs(15),lcrs(15)
      do 10 i=1,8
        ivrt(i)=0
10    continue
      do 20 i=1,15
        rcrs(i)=0
        lcrs(i)=0
20    continue
      i=1
30    n=0
40    n=n+1
      if (n .gt. 8) goto 60
      if (ivrt(n) .eq. 1) goto 40
      if (rcrs(i-n+8) .eq. 1) goto 40
      if (lcrs(i+n-1) .eq. 1) goto 40
        ipos(i)=n
        ivrt(n)=1
        rcrs(i-n+8)=1
        lcrs(i+n-1)=1
        i=i+1
      if (i .le. 8) goto 30
      write(*,50) (ipos(iy),iy=1,8)
50      format(' answer=',8i3)
-- More --^C……CTRL-Cを入力する

 バッチ処理を中止しますか <Y/N> y↵……バッチファイル実行時にSTOPまたはCTRL-Cを入力すると，メッセージを
A>■                                  表示してバッチ処理を中止するかどうか確認が求められる
```

Fig. 2.66　バッチ処理の実行と中止

2.6.2　パラメータを用いたバッチ処理

前項のバッチファイルでは，バッチファイル中に開発するプログラム名がすでにはいっていました．この方法ですと，別のプログラム，TEST2，TEST3……を開発するたびに，バッチファイルを書き直さなければいけません．

このような場合，バッチファイル中には，何か仮のファイル名を使っておいて，バッチファイルの実行時に，実際のファイル名を指定できると便利です．

MS-DOS ではこのようなときのために，バッチ処理用のパラメータが 10 個用意されています．%0，%1，……%9 の 10 個のパラメータを使ってバッチファイルを作っておき，実行時に各パラメータに対応する実際のファイル名を指定すると，自動的に置換えが行われます．

前項の例のバッチファイルを書き直したものを Fig. 2.67 に示します．ファイル名は〝DEVFORT．BAT〟です．

```
A>type devfort.bat ↵ ……〝DEVFORT．BAT〟を表示する
edlin %1.for ……EDLINでソースファイルを作成する
more < %1.for ……MOREコマンドでソースファイルの内容を確認する
b:for1 %1.for; ……FORTRANコンパイラpass one
b:pas2 ……FORTRANコンパイラpass two
b:link %1,%1,nul,b:fortran ……オブジェクトファイルとライブラリ〝FORTRAN．LIB〟とをリンクする
%1 ……作成したプログラムを実行する

A>■
```

Fig. 2.67　パラメータを用いたバッチファイル①

このバッチ処理では，%1 と TEST が対応しており，実行時に自動的に文字の置換えが行われます．置換え後の〝DEVFORT．BAT〟の内容は，先の〝DEVTEST〟と同じです．

この，バッチファイルで，〝TEST2〟を開発するときは，次のように入力すればよいことはいうまでもありません．

```
DEVFORT　TEST2 ↵
```

記号%0～%9 は，実行時に単純な文字の置換えで処理されますので，ファイル名に限らず，もっと汎用的に使うことができます．

Fig. 2.68 では，パラメータをドライブ名，ファイル名，スイッチ，FIND コマンドの文字列として用いています．

```
A>type test.bat ↵ ……〝TEST．BAT〟を表示する
dir %1 %2 ……ディレクトリの内容を表示する．%1，%2にはドライブ名，オプションなどがはいる
type %3 ……%3をタイプアウトする
find "%4" %3 > %5 ……ファイル%3の中から%4を含む行を検索してファイル%5に出力する

A>■
```

Fig. 2.68　パラメータを用いたバッチファイル②

このバッチファイル名を〝TEST．BAT〟とし，

```
TEST　A:　／W　ADDRESS　東京都　ADDTOKYO
```

と入力すると，次のようなコマンドを入力したときと同じ処理が行われます．

```
DIR A：／W
TYPE ADDRESS
FIND "東京都" ADDRESS > ADDTOKYO
```

2.6.3　バッチファイルの応用

バッチコマンドには簡単な条件判定や制御を行うコマンドがあるため，簡単なプログラムを記述することができます．

Fig. 2.69 に示したバッチファイルの内容は，複数のファイルを一度に指定できる DEL コマンドです．ファイル名は"DELALL．BAT"とします．ここではこのバッチファイルを例に，いくつかのバッチコマンドについて簡単に説明しておきます．BASIC になじみのある読者なら，リストを見るだけでほとんど理解できると思います．

```
A>type delall.bat
echo on ……ECHOをONにする．ECHO OFFだと次のREMコマンドが表示されない
rem *** sample batch file for command ***
echo off ……ECHOをOFFにする
:loop ……ラベル "loop"
if "%1"=="" goto end ……%1が空文字ならば "end" へ飛びバッチ処理を終了する
if not exist %1 goto next ……%1が存在しなければ "next" へ飛ぶ
del %1 ……%1を削除
:next ……ラベル "next"
shift ……コマンドラインに与えられたファイル名とパラメータとの関係を1つずつずらす．その結果%1には次のファイルが割り当てられる
goto loop……"loop" へ飛ぶ
:end ……ラベル "end"

A>■
```

Fig. 2.69　DELALL SHIFTコマンドを用いたバッチファイル

まず，1行目から3行目を見てください．1および3行目の ECHO はスクリーンへの表示を行うか否かの切換えスイッチです．ON で出力し，OFF で出力しなくなります．2行目の REM は単にメッセージを表示させるものです．1行目で ECHO を ON にしているのは REM 以降のメッセージを表示させるためで，メッセージを出力した後は出力があるとわずらわしいので出力を OFF にしています．

5，6行目の IF は条件判断を行います．5行目の IF では入力が空文字列であるか否かの判定を行い，6行目では入力されたファイル名が存在するか否かを判定しています．どちらも結果が真であれば GOTO の後ろに指定されたラベルに制御が移ります．

8および11行目がその飛び先のラベル名です．ラベル名の先頭にはコロンを付けなければなりません．9行目の SHIFT は，コマンドラインに与えられたファイル名を順番にパラメータ%1に割り当てる働きをします．

ここで示した例は非常に単純なものですが，コマンドの組合せによってはもっと複雑な処理を

行うことも可能です．第II部ではバッチコマンドの解説とともに実例も数多く掲載していますので，そちらも併せて参照してください．

2.6.4　AUTOEXEC．BAT

バッチファイルの特殊な形態として，自動実行バッチファイルと呼ばれるものがあります．このバッチファイルも基本的には他のバッチファイルと同様にファイルの中のコマンドを順次実行していくものですが，他のバッチファイルのようにユーザーのコマンド入力によって実行されるものではなく，MS-DOS のブート時に自動的に実行されるバッチファイルです．

なお，自動実行バッチファイルのファイル名は，かならず〝AUTOEXEC．BAT〟でなければなりません．

このバッチファイルは，つねに同じアプリケーションだけを利用するようなユーザーやシステムのブート時にかならず行うような処理の場合に効果的です．Fig. 2.70 では，マイクロソフト社の開発した作表言語 Multiplan を例に〝AUTOEXEC．BAT〟を作成しています．

```
A>type autoexec.bat ↵……ブート時に，日付と時刻を設定しMultiplanを実行するバッチファイル
date ┐
time ┘……〝AUTOEXEC．BAT〟が存在する場合，DATE，TIMEの両コマンドは自動的に実行されないのでこれらを
         〝AUTOEXEC．BAT〟にあらかじめ入れておく
mp ……Multiplanを起動する

A>■
```

Fig. 2．70　自動実行バッチファイル

2.7 コマンドと環境

MS-DOS にはプログラムの実行にかかわる環境という概念があります．これも UNIX から受け継いだ概念であり，プログラムが実行にあたって用いる変数にユーザーが自由に値をセットすることができるというものです．

この変数は〈環境変数〉と呼ばれ，〈環境変数〉に割り当てる値は SET コマンドで〝=〟で結んで指定します．この〝〈環境変数〉=〈文字列〉〟の集まりを MS-DOS では環境と呼んでいます．Fig. 2.71 は SET コマンドで環境を設定し，それを表示したものです．

```
A>set os=MS-DOS ↵……環境変数OSに文字列MS-DOSを割り当てる

A>set ↵……現在設定されている環境変数とその文字列を表示する
COMSPEC=A:¥COMMAND.COM ……〝COMMAND．COM〟を再ロードするときのパス名とコマンド・プロセッサ名
PATH=A:¥;A:¥BIN ……外部コマンドのサーチパス名
EPATH=a:¥bin ……アプリケーション・プログラムが使用する
OS=MS-DOS ……設定された

A>■
```

Fig. 2．71　SETコマンドによる環境の設定

環境に設定する値は，通常バッチファイルで用いられます．バッチファイルで〈環境変数〉を使う場合には，その前後に〝%〟を付けて〝%〈環境変数〉%〟と書きます．Fig. 2.72の例は，上でセットした環境をバッチファイルでメッセージとして出力したものです．

```
A>type environ.bat↵
echo off……ECHOをOFFにする
echo %os% is running. ……メッセージを表示する
        :……バッチファイル実行時には環境変数OSに設定されている文字列に置き換わる
A>environ ↵……バッチファイル〝ENVIRON．BAT〟を実行する

A>echo off
MS-DOS is running.
    :……環境変数OSに設定されている文字列
A>■
```

Fig. 2.72　バッチファイルにおける環境の利用

たとえばコマンド・プロセッサ(COMMAND．COM)も1つのプログラムであるため，ブート時にシステム側でその環境が設定されます．システムのブート直後の状態でSETコマンドを実行するとFig. 2.73のように表示されます．

```
A>set ↵……ブート直後にSETコマンドで現在設定されている環境変数を表示する
PATH=……外部コマンドのサーチパス(現在は何も設定されていない)
COMSPEC=¥COMMAND.COM ……〝COMMAND．COM〟を再ロードするときのパス名

A>■
```

Fig. 2.73　COMMAND．COMの環境

この状態では，COMSPEC(コマンド・プロセッサをロードしてくるディレクトリとそのファイル名)，PATH(コマンドのサーチパス)が表示されていますが，これはコマンド・プロセッサが使用している環境を表しています．

もう1つコマンド・プロセッサの環境として設定されるものにPROMPTがあります．MS-DOSのプロンプトを変更するにはPROMPTコマンドがありますが，SETコマンドでも同様にしてプロンプトを変更することができます．

```
A>set prompt=$g$n$l ↵……SETコマンドでプロンプトを〝>A<〟に変更する．〝PROMPT $g$n$l〟でも同じ

>A<
  :……プロンプトが変更された
```

Fig. 2.74　SETによるプロンプトの変更

3章 MS-DOSとソフトウェア開発

ソフトウェアの開発環境を提供することは，DOSの重要な機能の1つです．ソフトウェアの開発を行う場合，その生産性や保守性を高めるため，高級言語レベルでのプログラム作成を可能とする環境が必要となります．そのため，利用できる高級言語もDOSの機能の一部として考えることができます．

また，DOSの機能の1つとして，システムコールと呼ばれるサービスルーチンがあります．システムコールは，いくつかパラメータを与えるだけで，デバイスとの入出力，ディスクアクセスとファイル管理，メモリ管理といった機能を実現します．ユーザーは，システムコールを用いることにより，プログラムの開発を低レベルのデバイスコントロールを行う部分から作成せずに済むほか，よりコンパクトなプログラムを作成することができます．

DOS上でソフトウェアを開発するメリットは，このような働きをするシステムコールが利用できる点です．高級言語の中には言語レベルでシステムコールを利用できるものもありますが，多くの場合，機械語レベルで作成したプログラムとリンクしなければ，システムコールは利用できません．また，デバイスに密着していたり，高度に高速性が要求される場合のように，高級言語では記述しにくい，あるいはできないようなプログラムはアセンブラで開発することになります．

このように，ソフトウェアの開発は高級言語のみで行えるわけでなく，機械語レベルのプログラム開発を抜きにして考えることはできません．

本章では，実際のアセンブラ・プログラムの開発を例にとり，その開発手順とMS-DOSの各種ツールの働きを紹介します．

3.1 MS-DOSのプログラム開発用ユーティリティ

ソフトウェア開発などの際に有効な役割を果たすツールをユーティリティと呼びます．ユーティリティはDOSの一部となっており，プログラムを入力するのに使用するエディタ，プログラムを機械語にするアセンブラやコンパイラ，プログラムの実行状態を監視したり修正するデバッガなどがあります．

MS-DOSには標準として，マクロアセンブラでプログラム開発を行う際に必要な以下の7種類のユーティリティが提供されています．以下では個々のユーティリティについて簡単に紹介していきます．

EDLIN

ソースファイルの作成／編集を行います．スクリーン上で自由にテキストの編集が行えるスクリーンエデイタが利用できればそれに越したことはありませんが，ラインエディタであるEDLINでも十分その役割は果たすことができます．

MASM

MASMは8086／8088CPUのコードを生成するマクロ機能を持つアセンブラです．オブジェクトファイルは，マイクロソフト社の他の高級言語と互換性を持つため，各種の言語で作成したオブジェクトモジュールと組み合せてプログラムを作成することができます．

CREF

アセンブラで作られた，クロスリファレンス(拡張子〝．CRF〟)から，リスティングファイル(拡張子〝．REF〟)を作成します．クロスリファレンスを作成するとき，MASMはリスティングファイルに行番号を付けますが，この行番号を使って，すべてのシンボルがアセンブリ言語のソース・プログラムのどこで使用されているかを示したアスキーコード順のリストを作成します．

LIB

LIBはライブラリを管理するプログラムで，オブジェクトを必要に応じて削除／挿入します．そのほか，ライブラリファイルの整合性のチェック，PUBLICシンボル(他のオブジェクトモジュールから外部参照できるシンボル)のクロスリファレンスのリスティングの作成を行うことができます．

LINK

個々に開発したオブジェクトモジュールを1つにまとめ，必要なオブジェクトモジュールをライブラリから検索して結合し，後述するEXE形式のファイルを作成します．さらに，LINKはリロケーション情報とエラーメッセージのリストの作成も行うこともできます．オブジェクトファイルはリンクを行ってはじめて実行可能となります．

EXE2BIN

EXE形式のファイルをCOM形式のファイルに変換します．リロケーション項目がなく，実行開始アドレスがファイルの先頭であるといったEXE形式のファイルのヘッダがなくてもよいという条件が満たされれば，EXE形式のファイルをCOM形式のファイルに変換することができます．COM形式のファイルは作成が簡単で，EXE形式のファイルよりファイルサイズを小さくすることができ，ロード時間が短縮できます．

DEBUG

実行形式のファイルの，監視／制御を行い，デバッグのためのテスト環境を提供します．また，プログラムの修正を行ったり，DEBUG 自体でプログラムを作成することも可能です．

3.2 MS-DOS上でのプログラム開発

3.1 で述べた各種ユーティリティを使って，プログラム開発を行う際の概略を Fig. 3.1 に示します．

(1) EDLIN または MS-DOS 上で実行可能なエディタを用いて，アセンブリ言語や高級言語のソースファイルを作成します．

(2)アセンブリ言語や高級言語のソースファイルを，MASM でアセンブル，または高級言語のコンパイラでコンパイルを行い，〝．OBJ〞の拡張子を持つオブジェクトファイルを作成します．

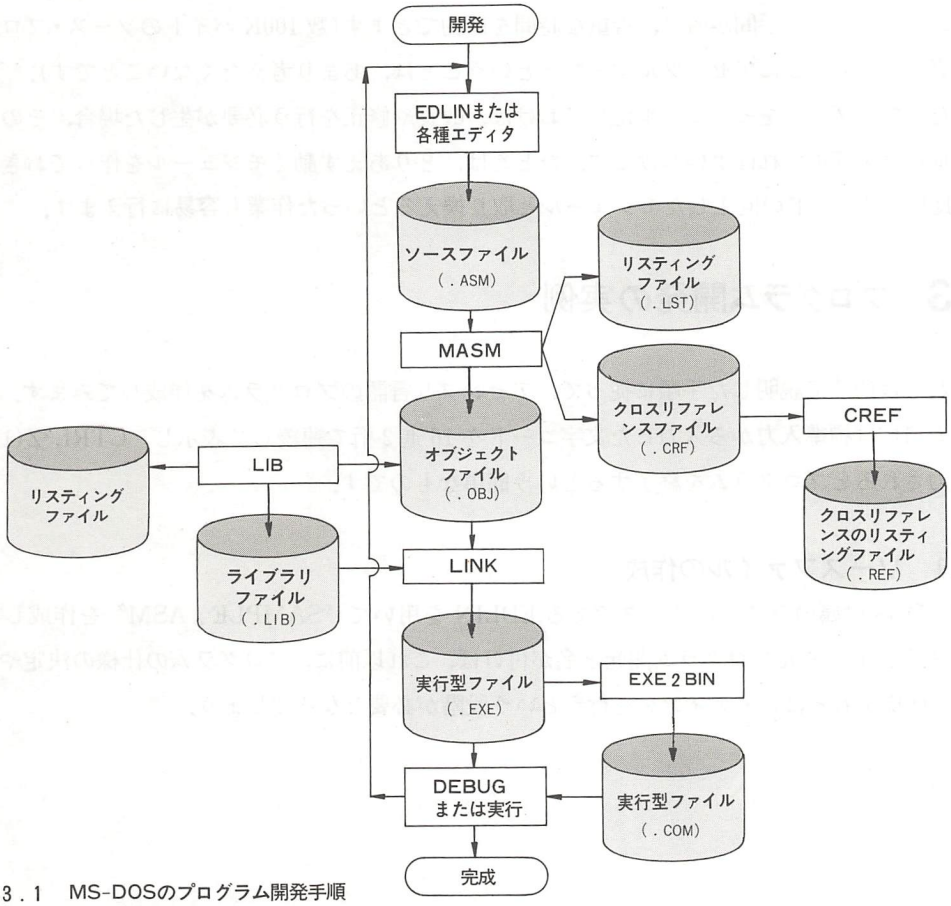


Fig. 3．1　MS-DOSのプログラム開発手順

(3) LINK を用いて1つまたはそれ以上のオブジェクトモジュールをリンクして，EXE 形式のファイルを作成します．このとき高級言語付属ライブラリやユーザーが LIB で作成したライブラリファイルを参照させることができます．さらに条件が許せば，EXE2BIN を用いて COM 形式に変換することができます．

(4) MS-DOS またはユーザーシステム上で，実行形式のファイルを動かしてみます．DEBUG の中から実行させる場合もあります．この実行結果が期待どおりであれば，プログラムの開発は終了です．エラーがあれば DEBUG を用いてプログラムの修正を行ったり，ソース・プログラムのレベルでの修正や，設計自体の修正を行う場合もあります．

アブソリュート・アセンブラに比べれば，この手順は LINK や LIB を利用するだけ複雑になっていますが，この機能はそれを補う十分な働きをします．巨大なプログラムではもちろん，小さなプログラムでも，モジュール化はプログラム開発において重要な技法といえます．一度作成してデバッグしたプログラムの単位となる部分は，オブジェクトもしくはライブラリとして保存しておけば，他のプログラムの作成にも利用できるので，ソース・プログラムの挿入，アセンブルやコンパイルをする手間が省け，貴重な時間を節約できます(数 100K バイトのソース・プログラムのデバッグのたびにアセンブルするなどということは，あまり考えたくないことです)．

また，プログラムをモジュール化しておけば，改良や修正を行う必要が生じた場合，そのモジュールだけを変更すればよいわけです．たとえば，とりあえず動くモジュールを作っておき，後で改良してスピードの向上したモジュールと取り換えるといった作業も容易に行えます．

3.3　プログラム開発の実例

それでは前節で説明した手順に従って，アセンブリ言語のプログラムを作成してみます．プログラム自体は標準入力から入力した文字コードを16進2桁で連続して表示し，CTRL-Z(1AH)が入力されるとプログラムを終了するという簡単なものです．

3.3.1　ソースファイルの作成

MS-DOS 付属のラインエディタである EDLIN を用いて "SAMPLE．ASM" を作成します(Fig. 3.2)．もちろんプログラム開発と名が付けば，これ以前に，プログラムの仕様の決定や，決定した仕様をもとにコーディングを行うという段階が必要となるでしょう．

```
A>edlin sample.asm
新しいファイルです
*i
     1:*;--------------------------------;
     2:*;        sample program          ;
     3:*;--------------------------------;
     4:*
     5:*        extrn   hexdisp:near,putch:near ……"HEXDISP"と"PUTCH"を外部参照として定義する
     6:*
     7:*cseg    segment byte public 'code'
     8:*        assume  cs:cseg,ds:cseg
     9:*sample  proc
    10:*
    11:*start:  mov     ah,7          }エコーバックなしの1文字入力
    12:*        int     21h           }
    13:*        cmp     al,1ah  ;=^Z  CTRL-Z(1AH)が入力されるとプログラムを終了する
    14:*        je      return
    15:*        call    hexdisp……ALレジスタの値を16進2桁で表示する
    16:*        mov     al,' '        }
    17:*        call    putch         }スペースを2つ空ける
    18:*        call    putch         }
    19:*        jmp     start……"start"に戻る
    20:*
    21:*return: int     20h……プログラムの終了
    22:*
    23:*sample  endp
    24:*cseg    ends
    25:*        end     sample
    26:*^Z
*e

A>■
```

Fig. 3．2　EDLIN実行結果

3.3.2　オブジェクトファイルとクロスリファレンスの出力

EDLINで作成した"SAMPLE．ASM"をMASMでアセンブルしてオブジェクトファイル"SAMPLE．OBJ"を作成します．また，クロスリファレンスファイル"SAMPLE．CRF"も作成しておいて，CREFを使って"SAMPLE．REF"の作成も行います(Fig. 3.3)．

```
A>dir sample

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥prog

SAMPLE   ASM       382  84-03-29  23:43……ソースファイル
       1 個のファイルがあります
  193536 バイトが使用可能です

A>masm sample,,sample,sample……アセンブルを行う．パラメータはソース，オブジェクト，リスティング，クロスリファレンスの順に指定する．この場合オブジェクトファイル名は省略したのでデフォルトの"SAMPLE．OBJ"となる．このときリスティングファイルおよびクロスリファレンスファイルも作成する
The Microsoft MACRO Assembler
Version 1.20, Copyright (C) Microsoft Inc. 1981,82,83

Warning Severe
Errors  Errors
0       0

A>cref sample;……クロスリファレンスファイル"SAMPLE．CRF"を入力とする．出力は";"を指定したのでデフォルトの"SAMPLE．REF"となる
```

```
A>dir sample

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥prog

SAMPLE   ASM       382  84-03-29  23:43
SAMPLE   OBJ       110  84-03-29  23:43 ……オブジェクトファイル
SAMPLE   LST      1505  84-03-29  23:43 ……リスティングファイル
SAMPLE   CRF       135  84-03-29  23:43 ……クロスリファレンスファイル
SAMPLE   REF       431  84-03-29  23:43 ……クロスリファレンスのリスティングファイル
       5 個のファイルがあります
  188416 バイトが使用可能です

A>type sample.lst
 The Microsoft MACRO Assembler                                  Page     1-1
                                                                03-29-84

 ┌──クロスリファレンスを作成したのでリスティングファイルに行番号が付けられる
 ↓  この行番号はクロスリファレンスで参照される
1                                       ;--------------------------------;
2                                       ;       sample program           ;
3                                       ;--------------------------------;
4
5                                               extrn   hexdisp:near,putch:near
6
7       0000                            cseg    segment byte public 'code'
8                                               assume  cs:cseg,ds:cseg
9       0000    ~                       sample  proc
10
11      0000    B4 07                   start:  mov     ah,7
12      0002    CD 21                           int     21h
13      0004    3C 1A                           cmp     al,1ah  ;=^Z
14      0006    74 0D                           je      return
15      0008    E8 0000 E                       call    hexdisp
16      000B    B0 20                           mov     al,' '
17      000D    E8 0000 E                       call    putch
18      0010    E8 0000 E                       call    putch
19      0013    EB EB                           jmp     start
20
21      0015    CD 20                   return: int     20h
22
23      0017                            sample  endp
24      0017                            cseg    ends
25                                              end     sample

 The Microsoft MACRO Assembler                                  Page     Symbols
                                -1
                                                                03-29-84


Segments and groups:

                N a m e                 Size    align   combine class

CSEG . . . . . . . . . . . . . .        0017    BYTE    PUBLIC  'CODE'

Symbols:

                N a m e                 Type    Value   Attr

HEXDISP. . . . . . . . . . . . .        L NEAR  0000            External
PUTCH. . . . . . . . . . . . . .        L NEAR  0000            External
RETURN . . . . . . . . . . . . .        L NEAR  0015    CSEG
SAMPLE . . . . . . . . . . . . .        N PROC  0000    CSEG    Length =0017
START. . . . . . . . . . . . . .        L NEAR  0000    CSEG

Warning Severe
Errors  Errors
0       0
```

の必要なものをライブラリから取り出し，組み込みます．

高級言語では，ほとんどの処理はライブラリのモジュールを呼び出して使用する形となっていますから，ライブラリの豊富さは使い勝手に直接影響します．また，高級言語にライブラリは必須ですから，コンパイルしたオブジェクトがリンク時にライブラリの指定を省略しても，自動的に必要なライブラリが参照されるようになっているものもあります．

この例ではオブジェクトファイル〝SAMPLE．OBJ〟と〝L1．LIB〟，〝L2．LIB〟の2つのライブラリファイルをリンクして〝SAMPLE．EXE〟を作成します(Fig. 3.5)．

```
A>link sample,,sample,l2 l1↵……ライブラリを〝L2．LIB〟,〝L1．LIB〟の順に指定する

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Warning: No STACK segment ……スタック・セグメントの指定がないのでエラーが出るが，実行には支障はない

Unresolved Externals:                    〕〝L1．LIB〟中の〝HEXDISP〟から外部参照される〝HALF〟と
                                         〕いうラベルが見つからず，エラーとなっている
HALF in file(s):
        A:L1.LIB(HEXDISP)

There were 3 errors detected.

A>■
```

Fig. 3．5　LINKエラー実行結果

HALFは〝L2．LIB〟内にあるはずなのにエラーが出ました．ここで，ライブラリを与える順序を変えてみます(Fig. 3.6)．

```
A>link sample,,sample,l1 l2↵…こんどはライブラリを〝L1．LIB〟,〝L2．LIB〟の順に指定する

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Warning: No STACK segment          〕スタック・セグメントを設定していないのでこのエラーが出る
                                   〕COMファイルを作成する場合は，無視してよい
There was 1 error detected.

A>■
```

Fig. 3．6　LINK実行結果

このように，リンク時にライブラリを与える順序に注意しないとエラーが発生する可能性があります．あるライブラリ内のモジュールが別のライブラリのモジュールを参照している場合，外部参照したモジュールは，それ以降に並べたライブラリにはいっているか，それより以前に参照されていなくてはならないのです．この例のように，必要なオブジェクトが前に指定されていてもリンクされない可能性があるので，複数のライブラリを扱う場合には注意が必要です．

3.3.5　MS-DOSの実行ファイル(EXE2BIN)

MS-DOSの実行可能なファイルには，〝.EXE〟と〝.COM〟の2種類の拡張子を持つものがあります．この2つは，8086CPUが実行可能な機械語の命令で構成されている点では同じですが，EXEファイルはヘッダ部分にセグメントのリロケーション情報やプログラムの開始アドレス，スタックのセグメントとオフセットなどといったプログラムに関する情報を持ち，LINKで作成したプログラムは拡張子に〝.EXE〟の付いたこの形式になっています．これに対しCOMファイルは，機械語のコードとデータのみを持っており，スタックの設定などが不用です．そのため，EXEファイルのヘッダ部分を取り除き，COMファイルを作成すればファイルサイズが小さくなるのでロード時間を短縮することができます．また，この場合の〝SAMPLE.EXE〟のようにスタック・セグメントが設定されていないためにこのままでは実行できないといったことも考えられます．このような場合，EXE2BINによってEXEファイルをCOMファイルに変換します．

このEXEファイルとCOMファイルの上で述べたような違いは，EXEファイルは8086CPUのアーキテクチャを忠実に反映したものであるのに対し，COMファイルが8080CPUのアーキテクチャを受け継いでいるところからきています．

EXEファイルは，その構造上セグメントのリロケートが必要となり，ヘッダにセグメントのリロケーション情報を用意します．ロードするときにそれらは再配置されますが，その他にも，ヘッダ内の情報からプログラムの開始アドレス，スタックのセグメントとオフセットが算出され，その値がCS:IPおよびSS:SPにセットされて，プログラムが実行されます．

COMファイルは，使用可能なメモリの最下位セグメント・アドレスにロードされ，すべてのセグメント・レジスタにプログラムがロードされたセグメント・アドレスを与えられて，オフセット100Hから実行されます．Fig. 3.7はEXEとCOMの2種類の実行可能なファイルがロード，実行されるときのメモリ配置を示したものです．

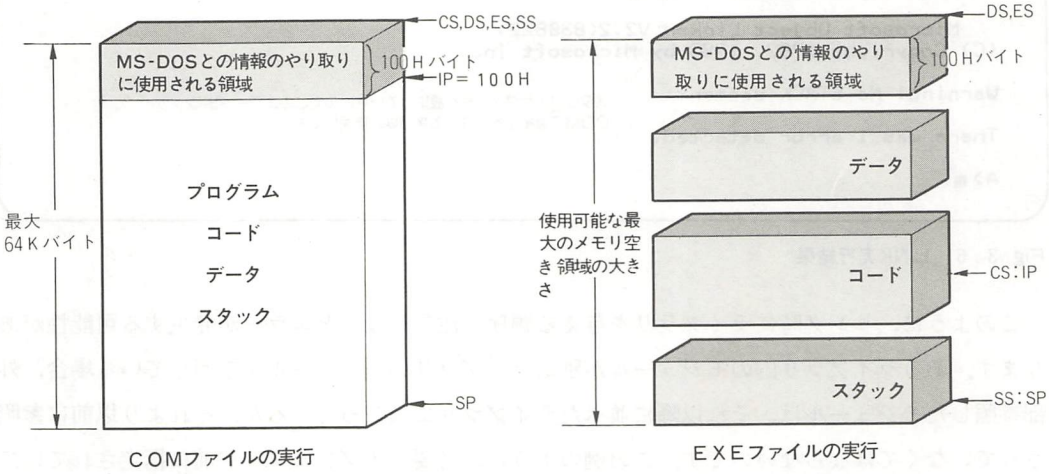


Fig. 3.7　COMファイル，EXEファイルの実行時のメモリ配置

プログラムがロードされたセグメント・アドレスのはじめには，DOS とプログラムとのパラメータの受渡しに利用される 256 バイトの領域が設けられていることがわかります．

それでは，"SAMPLE．EXE" から "SAMPLE．COM" を作成します(Fig. 3.8)．

```
A>exe2bin sample sample.com↵ ……"SAMPLE．EXE" からヘッダを取り除き
                                "SAMPLE．COM" を作成する
A>dir sample↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥prog

SAMPLE   ASM      382  84-03-29  23:43
SAMPLE   OBJ      110  84-03-29  23:43
SAMPLE   LST     1505  84-03-29  23:43
SAMPLE   CRF      135  84-03-29  23:43
SAMPLE   REF      431  84-03-29  23:43
SAMPLE   MAP      190  84-03-29  23:44
SAMPLE   EXE      577  84-03-29  23:44
SAMPLE   COM       65  84-03-29  23:44 ……"SAMPLE．COM" が作成された．ファイルの大きさは(EXEファイ
                                        ルの大きさ－５１２)バイトとなっている
        8 個のファイルがあります
   180224 バイトが使用可能です

A>■
```

Fig. 3．8　EXE2BIN実行結果

3.3.6　デバッグ作業

できあがった実行型ファイルを，ただ実行してみて動くか動かないか判断し，動かなければソースファイルの修正からやり直すというような方法をとっていたのでは，いつまでたってもプログラムが完成しないということにもなりかねません．そこで，トレースを行ったり，レジスタやメモリのチェックなどを行うデバッガの機能を利用してデバッグを行うことが必要となります．

デバッガといっても，ICE から BASIC のモニタ程度のものまでいろいろありますが，それほど高機能を必要としないのであれば，MS-DOS 付属の DEBUG で十分です．これには，小さな修正なら，アセンブル，リンクといった一連の手段を取ることなく，パッチを当ててしまうこともできます．

できあがった〝SAMPLE．COM〟をDEBUGで実行してみます(Fig. 3.9).

```
A>debug sample.com
-u
1635:0100 B407          MOV     AH,07
1635:0102 CD21          INT     21
1635:0104 3C1A          CMP     AL,1A
1635:0106 740D          JZ      0115
1635:0108 E80C00        CALL    0117
1635:010B B020          MOV     AL,20
1635:010D E82600        CALL    0136
1635:0110 E82300        CALL    0136
1635:0113 EBEB          JMP     0100
1635:0115 CD20          INT     20
1635:0117 50            PUSH    AX
1635:0118 51            PUSH    CX
1635:0119 B104          MOV     CL,04
1635:011B D2C8          ROR     AL,CL
1635:011D E80600        CALL    0126
-r                                  ファイルサイズは41H(65)バイト
AX=0000  BX=0000  CX=0041  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1635  ES=1635  SS=1635  CS=1635  IP=0100   NV UP DI PL NZ NA PO NC
1635:0100 B407          MOV     AH,07
-g 113 ……一度だけループを回して見る
66……〝f〟を入力したので，そのコード(66H)が16進2桁で表示された
AX=0720  BX=0000  CX=0041  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1635  ES=1635  SS=1635  CS=1635  IP=0113   NV UP DI PL NZ AC PE NC
1635:0113 EBEB          JMP     0100
-g……正常に動作しているので途中で止めずに実行してみる
0A  64  66  67  6E  64  6A  6B  41  78  3A  67  61
Program terminated normally ……CTRL-Z(1AH)を入力したのでプログラムが終了した
-q

A>sample < sample.asm ……実際にコマンドラインから起動する
3B  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  3B  0D  0A  3B  09  73  61  6D
2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D
70  6C  65  20  70  72  6F  67  72  61  6D  09  09  3B  0D  0A  3B  2D  2D  2D  2D  2D
2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D  2D
2D  2D  2D  2D  2D  2D  2D  2D  3B  0D  0A  0D  0A  09  65  78  74  72  6E  09  09  09
68  65  78  64  69  73  70  3A  6E  65  61  72  2C  70  75  74  63  68  3A  6E  65  78
65  61  72  0D  0A  0D  0A  63  73  65  67  09  73  65  67  6D  65  6E  74  09  09  09
62  79  74  65  20  70  75  62  6C  69  63  20  27  63  6F  64  65  27  0D  0A  09  09
09  61  73  73  75  6D  65  09  63  73  3A  63  73  65  67  2C  64  73  3A  63  73  65
73  65  67  0D  0A  73  61  6D  70  6C  65  09  09  70  72  6F  63  0D  0A  0D  0A  09
73  74  61  72  74  3A  09  6D  6F  76  09  61  68  2C  37  0D  0A  09  69  6E  74  21
74  09  32  31  68  0D  0A  09  63  6D  70  09  61  6C  2C  31  61  68  09  3B  3D  20
3D  5E  5A  0D  0A  09  6A  65  09  72  65  74  75  72  6E  0D  0A  09  63  61  6C  6C
6C  6C  09  68  65  78  64  69  73  70  0D  0A  09  6D  6F  76  09  61  6C  2C  27  27
27  20  27  0D  0A  09  63  61  6C  6C  09  70  75  74  63  68  0D  0A  09  63  61  6C
61  6C  6C  09  70  75  74  63  68  0D  0A  09  6A  6D  70  09  73  74  61  72  74  3A
74  0D  0A  0D  0A  72  65  74  75  72  6E  3A  09  69  6E  74  09  32  30  68  0D  0A
0D  0A  0D  0A  73  61  6D  70  6C  65  09  65  6E  64  0D  0A  63  73  65  67  09  65
67  09  65  6E  64  73  0D  0A  09  65  6E  64  09  73  61  6D  70  6C  65  0D  0A
A>■
```

Fig. 3．9　DEBUG実行結果

これで，〝SAMPLE．COM〟の完成となります．

第II部
コマンド・リファレンス

MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK
MS-DOS HANDBOOK

コマンド・リファレンスの形式

コマンド・リファレンスはFig.1.1で示す形式で説明されています．まず，各コマンドの見出し(①)に続いて**内部**あるいは**外部**の表示(②)があり，そのコマンドがMS-DOSの内部コマンドか外部コマンドかを示しています．その隣りの数字(③)はどのバージョンのMS-DOSで使用できるかを示しており，1.25と表示されているコマンドはMS-DOS ver. 1.25およびver. 2.0で使用可能なものを表し，2.0と表示されているコマンドはMS-DOS ver. 2.0でのみ使用できるものであることを表しています．なお，EDLINおよびDEBUGのコマンドには，これらの表示はありません．

次の**機能**(④)，および**書式**の欄(⑤)では，それぞれのコマンドの概略および正確な書式を示した後，**解説**の欄(⑥)ではそのコマンドの詳細な働きおよび注意すべき点について説明します．

最後の**サンプル・オペレーション**の欄(⑦)では，そのコマンドの働きをそのコマンドの使用例とともに説明します(EDLIN，DEBUGでは，実際のサンプル・オペレーションは前の解説中に含まれていますので，この欄は設けられていません)．

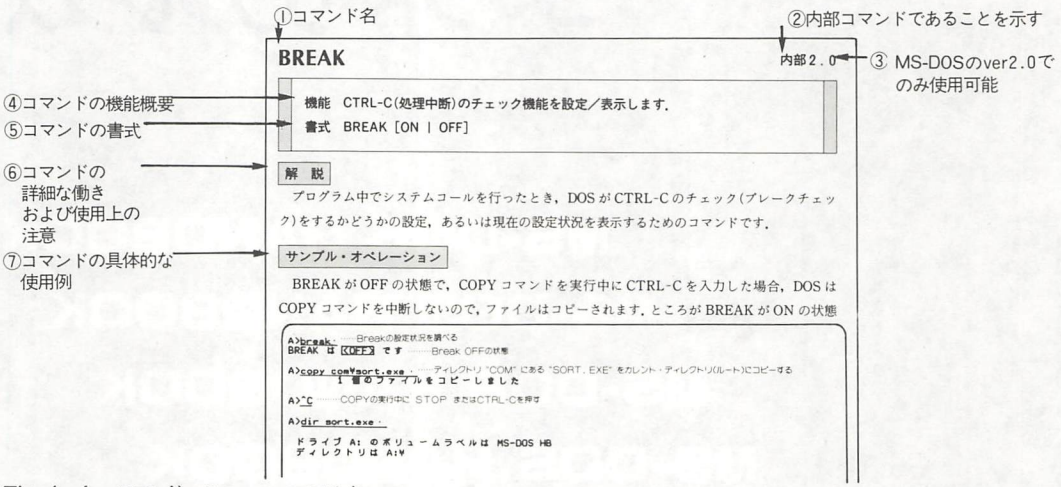


Fig. 1 . 1　コマンド・リファレンスの形式

コマンドの書式指定

第II部ではコマンドの書式に次のような表記法を用います．

[　]　(角形カッコ)…このカッコの中身は省略可能であり，必要に応じて入力する項目を示します．

〈 〉　(山形カッコ)…このカッコの中身はユーザーが入力するデータを示し，文字列が山形カッコで囲まれている場合には，それによって指定された項目を入力します．

{ }　(中カッコ)……このカッコは，そこに並べられた項目のうち，必要なものを選んで入力することを示します．角形カッコで囲まれていない限り，かならず1つは入力しなければなりません．

……　(省略記号)……必要に応じて何回も繰り返すことができる項目を示します．

サンプル・オペレーションについての注意

実際にDOSおよびEDLIN，DEBUGのコマンドのオペレーションでは数種類のディスクを用いていますが，システム・ディスクとして使用したディスクの最初のディレクトリ構造はFig. 1. 2のようになっており，これがオペレーションを行う際の基本的なディスクの状態を示しています．

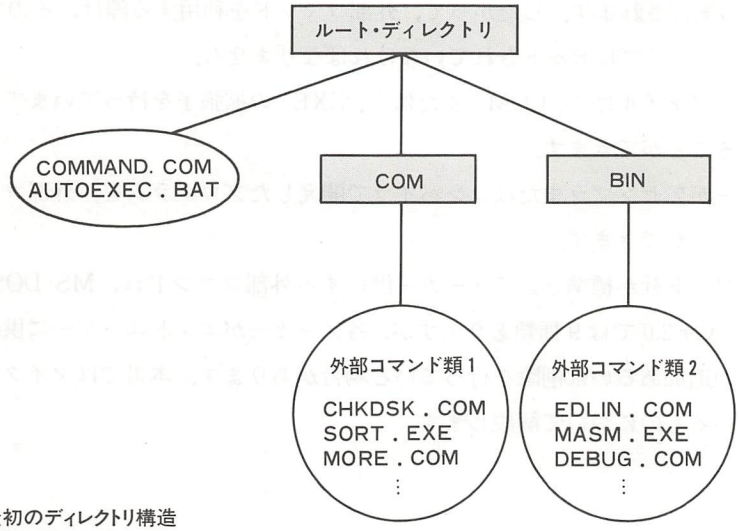


Fig. 1 . 2　最初のディレクトリ構造

外部コマンド類はサブ・ディレクトリ¥COM，および¥BINに分けて格納されています．そのため，外部コマンドの存在しないディレクトリからも外部コマンドを利用できるように，あらかじめPATHコマンドによって次のようにパスを設定しています．

```
A > PATH A : ¥ ; A : ¥COM ; A : ¥BIN ; ↵
```

外部コマンドの存在しないディレクトリで，外部コマンドを実行しているいくつかの例がありますが，パスを設定しておかないとこれらは実行することができません．

7章 MS-DOSコマンド

内部コマンド

内部コマンドは，MS-DOS ver. 1.25 で 9 種類，ver. 2.0 では 27 種類あります．いずれも頻繁に用いられる，サイズの比較的小さなプログラムであり，コマンド・プロセッサ〝COMMAND．COM〟に内蔵されています．したがって，MS-DOS のコマンドレベル(プロンプトが表示されている状態)であれば，いつでも使用することができます．

外部コマンド

外部コマンドは，使用頻度が低く，サイズの比較的大きなプログラムです．それぞれがファイルの形でディスクに収められており，コマンド名(ファイル名)が入力されるといったんメモリにロードされてから実行されます．したがって，外部コマンドを利用する際は，そのファイルがはいったディスクがドライブにセットされていなければなりません．

外部コマンドのファイルは〝．COM〟または〝．EXE〟の拡張子を持っていますが，実行時にはこれを省略することができます．

なお，ユーザーがアセンブラまたはコンパイラで開発したプログラムを，外部コマンドとまったく同等に扱うことができます．

注意：マイクロソフト社が標準としてメーカー供給する外部コマンドは，MS-DOS ver. 1.25 では 4 種類，ver 2.0 では 9 種類ありますが，各メーカーがエンドユーザーに供給する際は，コマンドの追加あるいは削除を行っている場合があります．本書ではマイクロソフト社が供給しているものについて解説します．

パラメータ

多くの MS-DOS のコマンドは，パス名やファイル名などのパラメータを伴いますが，以下にそのパラメータの書式について解説します．

〈d：〉　ドライブ名を表し，ドライブ数に従ってＡから順にアルファベット１文字で指定します．ドライブ名の直後には，かならずコロン〝：〟を付けます．

［例］　Ａ：　（ドライブＡ）
　　Ｂ：　（ドライブＢ）

〈ファイル名〉 ドライブ名および8文字以内の名前，3文字以内の拡張子を含めて次の書式で指定します．

［〈d：〉］〈名前〉［．〈拡張子〉］

［例］ COMMAND．COM
TEST
B：README．DOC

〈パス名〉 パス名は，目的のファイルに達するまでのディレクトリ名を連ねたもので，通貨記号〝¥〞で区切り，ドライブ名を含めて次の書式で指定します．

［〈d：〉］［¥］［〈ディレクトリ名〉¥］…［〈ディレクトリ名〉¥］〈ディレクトリ名〉

ここで，ドライブ名の次の通貨記号〝¥〞はルート・ディレクトリを表し，それを指定した場合，〈パス名〉はルート・ディレクトリからのパスを表し，それを省略した場合にはカレント・ディレクトリからのパスを表します．また，〈ディレクトリ名〉は親ディレクトリを表す〝..〞を使って指定することもできます．

［例］ A：¥BIN （ルート・ディレクトリのサブ・ディレクトリ〝BIN〞）
..¥WORK （親ディレクトリにあるサブ・ディレクトリ〝WORK〞）
B：BAT （ドライブBのカレント・ディレクトリにあるサブ・ディレクトリ〝BAT〞）

〈デバイス〉 MS-DOSでは，各種の周辺装置(デバイス)もファイルとして扱われており，いくつかのファイル名がデバイス名として予約されています．〈デバイス〉には以下に示すデバイス名を指定します．

AUX 補助入出力装置への入出力を指定します(入出力)．
CON キーボードからの入力，スクリーンへの出力を指定します(入出力)．
PRN プリンタへの出力を指定します(出力)．
NUL コマンド構文上，入出力のファイル名を要求されていても特にファイルを作りたくない場合に指定します．

バッチ・コマンド

バッチ・コマンドは，一般的なDOSのコマンドのように単独でも用いることができますが，特にバッチファイル中でその機能を発揮するものです．本書では，DOSのコマンドとは分けて解説しています．

BREAK

内部2.0

機能　CTRL-C(処理中断)のチェック機能を設定／表示します.
書式　BREAK [ON | OFF]

解　説

プログラム中でシステムコールを行ったとき，DOSがCTRL-Cのチェック(ブレークチェック)をするかどうかの設定，あるいは現在の設定状況を表示するためのコマンドです.

BREAK ONに設定すると，システムコールを行うたびにDOSがCTRL-Cのチェックを行います. BREAK OFFに設定すると，CTRL-Cのチェックは DOS側では行われず，プログラム側にゆだねられます.

なお，通常MS-DOSをブートしたときには，BREAK　OFFに設定されていますが，"CONFIG.SYS"ファイル(コンフィギュレーションファイル)の中で"BREAK=ON"を設定しておけば，ブート時にBREAK ONに設定されます.

ON，OFFを付けずにBREAKのみを入力すると，現在のBREAKの設定状況が表示されます.

システムコールによっては，BREAKのON, OFFにかかわらずCTRL-Cのチェックを行ってプログラムを自動的に中断してしまうものもありますし，またアプリケーション・プログラムによっては，CTRL-CによりDOSがプログラムを中断できないようにしているものもあります. したがってBREAK ONであってもプログラムを中断できなかったり，BREAK OFFであってもDOSがプログラムを中断してしまうことがあります. たとえば，EDLINはBREAK ONであってもCTRL-Cによって中断できませんが，TYPEコマンドはBREAKのON, OFFにかかわらずCTRL-Cによって中断できます.

サンプル・オペレーション

BREAKがOFFの状態で，COPYコマンドを実行中にCTRL-Cを入力した場合，DOSはCOPYコマンドを中断しないので，ファイルはコピーされます. ところがBREAKがONの状態では，DOSがCTRL-Cのチェックを行うため，もしCOPYコマンドの実行中にCTRL-Cが入力されれば，DOSはCOPYコマンドの実行を中断し，ファイルのコピーは行われません.

```
A>break ……Breakの設定状況を調べる
BREAK は <OFF> です ……Break OFFの状態

A>copy com¥sort.exe ……ディレクトリ "COM" にある "SORT．EXE" をカレント・ディレクトリ(ルート)にコピーする
        1 個のファイルをコピーしました

A>^C ……COPYの実行中に STOP またはCTRL-Cを押す

A>dir sort.exe

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

SORT      EXE      1360  83-09-04   23:54 ……Break OFFの状態のためCOPYコマンドは中止されずに "SORT．EXE"
                                              がコピーされている
        1 個のファイルがあります
   825344 バイトが使用可能です

A>break on ……Break ONに設定

A>break
BREAK は <ON> です

A>copy com¥more.com ……ディレクトリ"COM"にある"MORE．COM"をカレント・ディレクトリ(ルート)にコピーする
^C ……COPYの実行中に STOP またはCTRL-Cを押す

A>dir more.com

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

ファイルが見つかりません ……Break ONの状態のためCOPYコマンドは中止され "MORE．COM" はコピーされていない

A>break off

A>break
BREAK は <OFF> です

A>■
```

CHDIR(CD)

内部２．０

機能　カレント・ディレクトリを変更／表示します．
書式　CHDIR［〈パス名〉］
別表記　CD［〈パス名〉］

解　説

カレント・ディレクトリ(ユーザーが現在着目しているディレクトリ)を変更，あるいは表示するためのコマンドです．

〝CHDIR 〈パス名〉〟を入力すると，カレント・ディレクトリをパス名で指定されたディレクトリに変更します．カレント・ディレクトリを変更するということは，ユーザー側から見れば現在のディレクトリから他のディレクトリへ移るということを意味します．また，CHDIR のみを入力するとカレント・ディレクトリを表示します．

(参考：MKDIR，RMDIR)

サンプル・オペレーション

最初のディレクトリが，Fig. 1.3のようになっているものとしてCHDIRコマンドの実行例を示します．

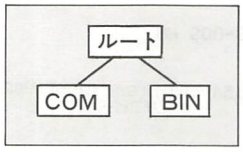


Fig. 1 . 3　ディレクトリの状態

```
A>dir ……カレント・ディレクトリ(ルート)の内容を表示する

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
KNJDIC   SYS     56320  83-09-20  14:22
BIN          <DIR>      84-02-22  20:32
COM          <DIR>      84-02-22  20:21
AUTOEXEC BAT        47  84-02-22  20:37
SORT     EXE      1360  83-09-04  23:54
       6 個のファイルがあります
   825344 バイトが使用可能です

A>chdir bin ……サブ・ディレクトリ "BIN" にカレント ・ディレクトリを変更する(サブ・ディレクトリ "BIN" に移る)

A>chdir ……カレント・ディレクトリを表示する
A:¥bin

A>mkdir subdir ……カレント・ディレクトリ "BIN" の下にサブ・ディレクトリ "SUBDIR" を作る

A>dir

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥bin

.            <DIR>      84-02-22  20:32
..           <DIR>      84-02-22  20:32
DEBUG    COM     12146  83-09-05   0:35
EDLIN    COM      9124  83-09-17  12:57
         EXE      1656  83-09-04  21:14
CREF     EXE     [illegible]  83-10-30  16:00
LINK     EXE     42624  [illegible]  15:40
MASM     EXE     80256  83-08-[illegible]
DUMP     COM      1670  83-09-17  19:14
SUBDIR       <DIR>      84-02-22  21:40 ……今作ったサブ・ディレクトリ "SUBDIR"
      11 個のファイルがあります
   824320 バイトが使用可能です

A>chdir subdir ……サブ・ディレクトリ "SUBDIR" に移る

A>dir

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥bin¥subdir

.            <DIR>      84-02-22  21:40 } サブ・ディレクトリを作ると自動的に作られる
..           <DIR>      84-02-22  21:40 } "." はカレント・ディレクトリ
       2 個のファイルがあります          ".." はその親ディレクトリ
   824320 バイトが使用可能です

A>chdir ..  ……親ディレクトリ ("BIN") に移る
          ――― 親ディレクトリを表す
A>chdir
A:¥bin …… BINに戻っている

A>chdir ..  親ディレクトリ(ルート)に移る
```

```
A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥
                ルートに戻っている
COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
BIN          <DIR>     84-02-22  20:32
COM          <DIR>     84-02-22  20:21
AUTOEXEC BAT       47  84-02-22  20:37
SORT     EXE     1360  83-09-04  23:54
       6 個のファイルがあります
   824320 バイトが使用可能です

A>cd bin¥subdir ↵……"BIN" のサブ・ディレクトリ "SUBDIR" に移る  CDはCHDIRの別表記

A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥bin¥subdir

.            <DIR>     84-02-22  21:40
..           <DIR>     84-02-22  21:40
       2 個のファイルがあります
   824320 バイトが使用可能です

A>cd ¥↵ ……ルートに戻る

A>■
```

CHKDSK

外部 1 .25

機能　指定されたドライブのディレクトリと FAT を調べ，ディスクの状態を報告します．

書式　CHKDSK [〈d : 〉] [〈ファイル名〉] [／F] [／V]

解　説

ディスク上に分散して記録されているファイルの一部が欠落していないか，ディスクの一部が二重に使われていないか，ディスクの空き領域がどの程度残っているかなどがチェックされます．

CHKDSK コマンドには／F と／V の 2 つのスイッチがあります．

／F スイッチを指定すると，ディスクにエラーが発見された場合，そのディスクを修復することができます．CHKDSK コマンドの実行によって発見されたディスクのエラーを，勝手に修復しては不都合が生じる場合があるため，この／F スイッチを指定しない限りディスクの修復は行われません．

／V スイッチを指定すると，ディスクに登録されているファイル名がすべて表示されます．これはファイルやディレクトリを多く作りすぎて，目的のファイルがどのディレクトリにあったか，あるいはどんなファイルがそのディスクに登録されているかを確認する場合に用いることができます．

また，〈ファイル名〉を指定すると，そのファイルのディスク上での割当て状況が表示されます．ファイル名にはワイルドカードも使用できます．

サンプル・オペレーション

CHKDSKコマンドを用いて，いくつかのディスクをチェックした例を示します．最後の例では発見されたエラーが／Fスイッチによって修復されています．

```
A>chkdsk a:↵……ドライブAのディスクの状態をチェックする
ディスク MS-DOS HB    は 1984-02-22 15:16 に作成されました

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     3072 バイト : 3 個のディレクトリ
   363520 バイト : 32 個のユーザーファイル
   824320 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   183808 バイト : 使用可能メモリ


A>chkdsk b: /v↵…… ／Vスイッチを付けてドライブBのディスクの状態をチェックする
ディスク MSDOSCOM     は 1984-02-22 21:27 に作成されました
ディレクトリ B:
      B:¥IO.SYS
      B:¥MSDOS.SYS
      B:¥COMMAND.COM
      B:¥MSDOSCOM
ディレクトリ B:¥HC
      B:¥HC¥BREAK.HC
      B:¥HC¥CHDIR.HC

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     1024 バイト : 1 個のディレクトリ
    22528 バイト : 3 個のユーザーファイル
  1167360 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   183808 バイト : 使用可能メモリ

A>chkdsk command.com↵……ファイル名 "COMMAND．COM" を指定してCHKDSKコマンドを実行する
ディスク MS-DOS HB    は 1984-02-22 15:16 に作成されました

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     8192 バイト : 8 個のディレクトリ
   414720 バイト : 56 個のユーザーファイル
   768000 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   183840 バイト : 使用可能メモリ

指定されたすべてのファイルは連続しています……指定したファイルに関する情報が表示される

A>chdir hc↵……ドライブAのディスケットを交換し，サブ・ディレクトリHCに移る

A>chkdsk *.*↵……すべてのファイルを指定してCHKDSKコマンドを実行する
ディスク MSDOSCOM     は 1984-02-22 21:27 に作成されました

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     4096 バイト : 3 個のディレクトリ
   126976 バイト : 38 個のユーザーファイル
  1059840 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   183872 バイト : 使用可能メモリ


A:¥hc¥CHKDSK.HC
  は 2 個の不連続ブロックを含みます
A:¥hc¥RECOVER.HC
  は 2 個の不連続ブロックを含みます
                    } 不連続ブロックを含むファイルが表示される
```

```
A>chkdsk b: /f ……ドライブBのディスケットを交換し／Fスイッチを付けてCHKDSKコマンドを実行する
ディスク MS-DOS HB    は 1984-02-22 15:16 に作成されました

1 チェイン中 1 個の破損クラスタが発見されました
破損チェインをファイルに変換しますか<Y/N>? y ……"y" を入力して破損チェインをファイルに変換する

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     8192 バイト : 8 個のディレクトリ
   414720 バイト : 56 個のユーザーファイル
     1024 バイト : 1 個の修復したファイル
   766976 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   183840 バイト : 使用可能メモリ


A>dir b:

 ドライブ B: のボリュームラベルは MS-DOS HB
 ディレクトリは B:¥

COMMAND  COM     17190  83-09-19  16:19
SYS      COM     56320  83-09-20  14:22
                        84-02-22  20:32
OPR                               22  20:21
OPR      TXT
SUBDIR1      <DIR>      84-02-
CAL          <DIR>      84-02-23   1:40
TXT          <DIR>      84-02-23   5:07
LANGUAGE TXT       167  84-02-23   5:38
FILE0000 CHK      1024 ……ファイルに変換された破損チェイン
      14 個のファイルがあります
  766976 バイトが使用可能です

A>■
```

CLS

内部2.0

機能　スクリーンを消去します.

書式　CLS

解　説

パラメータはいっさいありません. CLS コマンドはスクリーンへ ANSI エスケープ・シーケンス "ESC [2 J" を送ります.

(参考：エスケープ・シーケンス)

サンプル・オペレーション

CLS コマンドを入力すると，それまでスクリーン上にあった文字が消去され，その後プロンプトが表示されます.

```
A>cls ……画面をクリアする
A>■
```

COPY

内部1.25

機能　ファイル単位，またはディレクトリ単位でファイルのコピーを作ります．またファイルをコピーする際，いくつかのファイルを連結してコピーすることもできます．

書式　COPY［／A］［／B］{〈パス名1〉|〈ファイル名1〉}［／A］［／B］［{〈d：〉|〈パス名2〉|〈ファイル名2〉}］［／A］［／B］［／V］

解　説

COPYコマンドは第1パラメータ{〈パス名1〉|〈ファイル名1〉}で指定されたファイルを第2パラメータ{〈d：〉|〈パス名2〉|〈ファイル名2〉}にコピーします．また第1パラメータを〝＋〟でつなぐことによって，それらのファイルを連結してコピーすることもできます．

パラメータの指定方法とCOPYコマンドの実行結果には，Table 1.1のような関係があります．

Table 1.1　COPYコマンドのパラメータ指定と実行結果

パラメータの指定			COPYコマンド実行結果
第1パラメータ（ソース）	〈ファイル名〉		指定されたファイルがコピーされる
	〈パス名〉		指定されたディレクトリのすべてのファイルがコピーされる
第2パラメータ（ディスティネーション）	省略		カレント・ドライブのカレント・ディレクトリに第1パラメータで指定されたファイルが個別にコピーされる
	〈d：〉		〈d：〉で指定されたドライブのカレント・ディレクトリに第1パラメータで指定されたファイルが個別にコピーされる
	〈パス名〉		〈パス名〉で指定されたディレクトリに第1パラメータで指定されたファイルが個別にコピーされる
	〈ファイル名〉	第1パラメータ〈ファイル名〉	第1パラメータで指定されたファイルが第2パラメータで指定されたファイルにコピーされる
		第1パラメータ〈パス名〉	第1パラメータの〈パス名〉で指定されたディレクトリのすべてのファイルが連結されて，第2パラメータで指定されたファイルにコピーされる

COPYコマンドには／A，／Bおよび／Vの3つのスイッチがあり，／A，／Bスイッチは EOF（1AH：アスキーファイルの終わり）を無視するかどうかを指定するもので，／Vスイッチはベリファイを指定するものです．

／A,／Bスイッチはそれぞれアスキーモード，バイナリモードでコピーを行うことを指定します．第1パラメータで指定するか第2パラメータで指定するかによって意味が異なり，その関係はTable 1.2のようになります．なお，これらのスイッチが省略された場合，ファイルの連結を行う際にはアスキーモードで，そうでないときにはバイナリモードでコピーが行われます．

Table 1.2 ／A,／Bスイッチ

スイッチ	第1パラメータ(ソース)	第2パラメータ(ディスティネーション)
／A (アスキーモード)	ソースファイル中でＥＯＦが発見されたらＥＯＦとそれ以降のデータをディスティネーションに送らない	EOFをファイルの最後に書き込む
／B (バイナリモード)	ディレクトリに登録されているファイルサイズに従ってファイルの全データをディスティネーションに送る	EOFは書き込まない

／Vスイッチを指定すると，コピー終了後にディスクに正しく書き込まれたかどうかをチェック(ベリファイ)します．この場合コピーを完全に行うことができますが，処理時間がそれだけ余計にかかります．

COPYコマンドのパラメータをファイル名の代わりにデバイス名(CON，PRNなど)を指定すると，各デバイスとファイル，あるいはデバイス間でデータの転送を行うことができます．ただし，COPYコマンドのパラメータにAUXを指定した場合，うまくデータを転送できないことがあるので注意が必要です．

サンプル・オペレーション

COPYコマンドを使ってファイルをコピーします．またソース(第1パラメータ)にCON(キーボード)を指定して，キーボードから入力したデータをファイルにしてみます．

```
A>dir↵
 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
BIN          <DIR>      84-02-22  20:32
COM          <DIR>      84-02-22  20:21
AUTOEXEC BAT       61   84-02-25  17:22
CONFIG   SYS       70   84-02-28  20:33
FUNC         <DIR>      84-02-22  22:04
SUBDIR1      <DIR>      84-02-23   1:19
CAL          <DIR>      84-02-23   1:40
TXT          <DIR>      84-02-23   5:07
ASMWORK      <DIR>      84-02-25   7:22
       11 個のファイルがあります
   751616 バイトが使用可能です
```

```
A>copy a:*.* b:↵
A:COMMAND.COM
A:KNJDIC.SYS
A:AUTOEXEC.BAT
A:CONFIG.SYS
        4 個のファイルをコピーしました

A>copy bin¥edlin.com com¥ed.com /v ↵
        1 個のファイルをコピーしました

A>dir com¥ed.com↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥com

ED       COM      9124  83-09-17  12:57
        1 個のファイルがあります
   742400 バイトが使用可能です

A>dir func¥*.asm↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

HEXOUT   ASM       529  84-03-08   6:27
FUNC-45  ASM       896  84-02-16  18:31
FUNC-25  ASM       768  84-02-15  17:28
AUXIN    ASM       185  84-02-19  21:13
        4 個のファイルがあります
   742400 バイトが使用可能です

A>copy func¥hexout.asm+func¥auxin.asm b:↵
func¥HEXOUT.ASM
func¥AUXIN.ASM
        1 個のファイルをコピーしました

A>dir b:↵

 ドライブ B: のボリュームラベルは MS-DOS V211
 ディレクトリは B:¥

COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
AUTOEXEC BAT       61  84-02-25  17:22
CONFIG   SYS       70  84-02-28  20:33
HEXOUT   ASM      713  84-03-20  20:37
        5 個のファイルがあります
  1114112 バイトが使用可能です

A>mkdir b:work↵

A>copy a:asmwork b:work↵
A:asmwork¥FUNC-00.ASM
A:asmwork¥FUNC-0B.LST
        9 個のファイルをコピーしました

A>dir b:work↵

 ドライブ B: のボリュームラベルは MS-DOS V211
 ディレクトリは B:¥work

         <DIR>      84-03-20  20:42
         <DIR>      84-03-20  20:42
                    84-02-20  21:54
                    84-02-20  21:55
ASM      HIS
FUNC-06  LST        1375  84-02-
FUNC-0B  LST        3017  84-02-25
        11 個のファイルがあります
  1097728 バイトが使用可能です

A>copy a:asmwork b:work1↵
A:asmwork¥FUNC-00.ASM
A:asmwork¥FUNC-03.ASM
A:asmwork¥FUNC-06.ASM
```

……ドライブAのディスクのカレント・ディレクトリ(ルート)にあるすべてのファイルをドライブBのカレント・ディレクトリへコピーする

コピーは個別に行われ，コピーされたファイルのファイル名が表示される

……サブ・ディレクトリBINにある"EDLIN．COM を／V スイッチを付けてベリファイを指定して，サブ・ディレクトリ"COM"に"ED．COM"の名前でコピーする

……コピーされた

……サブ・ディレクトリFUNCにある "HEXOUT．ASM" と "AUXIN．ASM" を連結してドライブBにコピーする．その際コピーされたファイル名は "HEXOUT．ASM" となる

"copyA：＊．＊　B：" でコピーされた分

……"HEXOUT．ASM" と "AUXIN．ASM" を連結したファイル

……サブ・ディレクトリ "ASMWORK" のすべてのファイルをドライブBのサブ・ディレクトリ "WORK" にコピーする

コピーは個別に行われる

コピーされた

……サブ・ディレクトリ"ASMWORK"のすべてのファイルをドライブBの"WORK 1"にコピーする．この場合 "WORK 1" はサブ・ディレクトリではなくファイルなので，すべてのファイルが連結されてファイル "WORK 1" にコピーされる

```
A:asmwork¥FUNC-06.LST
A:asmwork¥FUNC-0B.LST
        1 個のファイルをコピーしました

A>dir b:work*

 ドライブ B: のボリュームラベルは MS-DOS V211
 ディレクトリは B:¥

WORK         <DIR>      84-03-20  20:42
WORK1            10348  84-03-20  20:43 ……コピーされている
       2 個のファイルがあります
  1086464 バイトが使用可能です

A>dir b:work

 ドライブ B: のボリュームラベルは MS-DOS V211
 ディレクトリは B:¥work

             <DIR>      84-03-20  20:42
..           <DIR>      84-03-20  20:42
FUNC-06  ASM       768  84-02-20  21:54
FUNC-0B  ASM           84-02-20  21:55
FUNC-00  LST               ..    21:56
ASM      HIS        42  8..
FUNC-06  LST      1375  84-02-25
FUNC-0B  LST      3017  84-02-25   8:21
       11 個のファイルがあります
  1086464 バイトが使用可能です

A>copy con samp.txt ……キーボードからの入力をファイルにコピーする
COPY CON <ファイル名>
を実行すると，短いファイルが簡単に作れます。
^Z ……CTRL-Z を入力するとコマンドレベルに戻る
        1 個のファイルをコピーしました

A>type samp.txt
COPY CON <ファイル名>
を実行すると，短いファイルが簡単に作れます。

A>dump test ……ドライブAのディスケットを入れ換え，"TEST" をダンプする

Dump Version 2.0

                                                  1AHアスキーファイルの終わりを表す(EOF)．"TEST" はアスキーファイルではなく，バイナリファイルなので途中にEOFがはさまれている
00000000  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
00000010  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
00000020  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
00000030  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?

A>dump config.sys ……"CONFIG.SYS" をダンプする

Dump Version 2.0


00000000  66 69 6C 65 73 3D 32 30-0D 0A 62 75 66 66 65 72   files=20..buffer
00000010  73 3D 32 30 0D 0A 62 72-65 61 6B 3D 6F 6E 0D 0A   s=20..break=on..
00000020  1A 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
          └ EOF "CONFIG.SYS" はアスキーファイルなのでEOFの後はダミー
A>copy test+config.sys test1 ……"TEST" と "CONFIG.SYS" を連結してファイル "TEST1" にコピーする
TEST
CONFIG.SYS
        1 個のファイルをコピーしました

A>dump test1 ……コピーした"TEST1" をダンプする

Dump Version 2.0
                                  ファイルを連結するとアスキーモードでコピーが行われ，アスキーモードではEOF（1AH）とそれ以降はコピーされないので "TEST" が19Hまでしかコピーされていない
00000000  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
00000010  10 11 12 13 14 15 16 17-18 19 66 69 6C 65 73 3D   ..........files=
00000020  32 30 0D 0A 62 75 66 66-65 72 73 3D 32 30 0D 0A   20..buffers=20..
00000030  62 72 65 61 6B 3D 6F 6E-0D 0A 1A 00 00 00 00 00   break=on........

A>copy test/b+config.sys/a test2/a ……"TEST" をバイナリ "CONFIG.SYS" をアスキーモードにして受け側の "TEST2" をアスキーモードしてコピーする
TEST                  └"CONFIG.SYS" をアスキーモードにして最後の1A
CONFIG.SYS     Hを送らないで，受け側の "TEST2" で1AHを付加する
        1 個のファイルをコピーしました
```

すると，"TEST" は1AHがあってもそれは無視されファイルの全内容がコピーされるのでうまく連結を行うことができる

```
A>dump test2

Dump Version 2.0

                                                        ┌"TEST" がすべてコピーされている
00000000  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
00000010  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
00000020  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
00000030  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
00000040  66 69 6C 65 73 3D 32 30-0D 0A 62 75 66 66 65 72   files=20..buffer
00000050  73 3D 32 30 0D 0A 62 72-65 61 6B 3D 6F 6E 0D 0A   s=20..break=on..
00000060  1A 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

A>copy config.sys/a conf1/b ·····送り側で1AH以降を送らずに，また，受け側で1AHを付加しないでコピー
        1 個のファイルをコピーしました   すると最後の1AHを取り除くことができる

A>dump conf1

Dump Version 2.0

00000000  66 69 6C 65 73 3D 32 30-0D 0A 62 75 66 66 65 72   files=20..buffer
00000010  73 3D 32 30 0D 0A 62 72-65 61 6B 3D 6F 6E 0D 0A   s=20..break=on..
                                                  └1AHが除かれている
A>copy config.sys/b conf2/a ·····今度は／Aと／Bをそれぞれ逆にすると送り側で最後の1AHまで送り，受
        1 個のファイルをコピーしました   け側で1AHを付加しているので1AHがもう1つよけいに付加される

A>dump conf2

Dump Version 2.0

00000000  66 69 6C 65 73 3D 32 30-0D 0A 62 75 66 66 65 72   files=20..buffer
00000010  73 3D 32 30 0D 0A 62 72-65 61 6B 3D 6F 6E 0D 0A   s=20..break=on..
00000020  1A 1A 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
     └よけいに付加された1AH
A>dir conf*.*

 ドライブ A: のボリュームラベルは PC-9801
 ディレクトリは A:¥

CONFIG   SYS        33  84-03-20  22:24 ……もとのファイル
CONF1               32  84-03-20  22:24 ……最後の1AHを取り除いたファイル
CONF2               34  84-03-20  22:24 ……最後にもう1つ1AHを付加したファイル
       3 個のファイルがあります
  791552 バイトが使用可能です

A>■
```

CTTY

内部2.0

機能　コマンドを入出力するデバイスを変更します.

書式　CTTY [¥] 〈デバイス名〉

解　説

コマンドを入出力するデバイスを,〈デバイス名〉で指定されるデバイスに変更します.通常MS-DOSは，コマンドおよびその実行結果などの入出力にコンソール(キーボード，スクリーン)を用いますが，これをRS-232Cなどの補助入出力装置につながれたターミナルなどに変更することができます．このとき，

A > CTTY ¥AUX ↵

とすると，これ以降のコマンドの入出力はターミナル側からのみ行われるようになり，本体側のスクリーンには何も表示されず，またキーボードからの入力もその時点では受け付けられなくなります．なお，再びコマンドの入出力をコンソールに戻すには，ターミナル側から

A > CTTY ¥CON ↵

と入力します．ここで注意しなければならないことは，〝CTTY ¥AUX〟によってコマンドを入出力するデバイスを変更しても，デバイス名 CON はターミナル側のコンソールを指すのではなく，やはり本体側のコンソールを指すということです．

MS-DOS が標準でサポートしているデバイスは，このほかにも PRN，NUL などがありますが，これらを CTTY コマンドの〈デバイス名〉に指定しても，入出力双方に使えるデバイスではないためまったく無意味です．

また，CTTY コマンドは，コマンドの入出力用にデバイス・ドライバを自作したときなど，そのドライバにコマンド入出力の制御を移すために使うことができます．

(参考：デバイス・ドライバ)

サンプル・オペレーション

MS-DOS が動作しているホスト・コンピュータ(ここでは NEC PC-9801)に RS-232 C ポートを介してターミナル(PC-8201)をつなぎ，ホスト側から〝CTTY ¥AUX〟を入力したときの実行例を示します．

RS-232 C を利用するためには，まずデータの転送速度，ビット長などの各パラメータを設定し，次に RS-232 C ポートを初期化しなければなりません．この操作を行うコマンドは機種によって異なりますが，PC-9801 用の MS-DOS では SPEED コマンドがこれにあたります．RS-232 C ポートの初期化を行わないで〝CTTY ¥AUX〟を実行すると，ターミナル側からもホスト側からもコマンドが入力できなくなりますから注意が必要です．また同様にターミナル側もパラメータの設定およびポートの初期化を行わなければなりません．このとき，ターミナル側で MS-DOS が動作している必要はまったくありませんが，漢字が正常に表示されない場合があります．

以上の準備が終わった後にホスト側から

A > CTTY ¥AUX ↵

と入力すると，ターミナル側にプロンプト

A >

が表示されて，コマンドが入力できるようになります．

```
A>speed rs232c-0  9600 bits-8 parity-none stop-1 xon ↵……RS-232Cのパラメータを設定し，ポートを初期化する
A>ctty ¥aux ↵……コマンドを入出力するデバイスをAUX（RS-232C）に変更する．これ以後ターミナル側から "CTTY¥CON" が入力されるまで本体側のモニタには何も表示されない．また，キーボードからの入力もそれまでキーバッファにたまるだけで何のコマンドも入力できなくなる


ターミナル側のコンソール入出力
A>
A>dir ↵……ターミナルからコマンドを入力すると，ターミナルのスクリーンにその結果が表示される

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
BIN          <DIR>      84-02-22  20:32
COM          <DIR>      84-02-22  20:21
AUTOEXEC BAT       47   84-02-22  20:37
SORT     EXE     1360   83-09-04  23:54
FUNC         <DIR>      84-02-22  22:04
        7 個のファイルがあります
   806912 バイトが使用可能です

A>chdir func↵

A>type auxin.asm↵

cseg    segment
        assume  cs:cseg
;
        org     100h
;
loop:   mov     ah,3
        int     21h
        cmp     al,1ah
        je      return
        mov     ah,2
        mov     dl,al
        int     21h
        jmp     loop
;
return: int     20h

cseg    ends

        end

A>ctty ¥con ↵……コマンドを入出力するデバイスをCON(本体側)に変更する
```

DATE

内部 1.25

機能　システムが管理する日付を表示／設定します．

書式　DATE [<yy> - <mm> - <dd>]

解　説

システムが管理する日付を表示または設定するためのコマンドです．MS-DOS は，ファイルを作成または更新した際に，その日付と時刻をディレクトリに記録します．DATE コマンドはその日付を管理するためのコマンドで，MS-DOS がブートされたときに〝AUTOEXEC．BAT〟が存

在しない場合，自動的に起動されるコマンドの１つです．

A > DATE ↵

と入力すると，現在の日付を表示して新しい日付の入力待ちになります．このとき表示されている日付を変更する必要がない場合は，そのままリターンキーを入力します．日付を変更する必要がある場合には，年，月，日をハイフン〝-〟またはスラッシュ〝／〟で区切りながら，数字で入力します．

またDATEコマンドに直接日付を付けて入力することもできます．たとえば，

A > DATE 1984-5-22 ↵

と入力すると日付が1984年5月22日に設定されます．各数字の範囲は次のとおりです．

〈yy〉　：1980～2079（年）（1900年代の場合，先頭の19は省略可）
〈mm〉：1～12　　（月）　（〈yy〉の範囲は機種によって異なります．この例はPC-9801の場合）
〈dd〉　：1～31　　（日）

なお，日付が変わったときの日付の更新，曜日，月の大小，うるう年に関する処理はMS-DOSが自動的に行います．

DATEおよびTIMEコマンドは，MS-DOSブート時に自動的に実行されるものですが，〝AUTOEXEC．BAT〟が存在している場合は，この２つのコマンドは実行されません．ですからクロック・デバイス（バッテリーバックアップされている内部時計）を持たない機種で〝AUTOEXEC．BAT〟を使用する場合には，これらのコマンドをあらかじめ〝AUTOEXEC．BAT〟ファイルに入れておくとよいでしょう．

MS-DOSには英語版と日本語版とがありますが，英語版の場合年，月，日の順序が〈mm〉-〈dd〉-〈yy〉（月，日，年）になっていますので，DATEコマンドを使用する際には注意が必要です．

（参考：TIME）

サンプル・オペレーション

DATEコマンドを用いて日付を設定した例を示します．

```
A>date 1984-01-01↵……日付を1984年１月１日に設定する

A>date↵……DATAのみを入力すると現在の日付を表示し日付の入力待ちになる
現在の日付は 1984-01-01（日）です
日付を入力してください：1984-02-23↵……1984年２月23日に設定する

A>■
```

DEL(ERASE)

内部 1 .25

機能　　指定されたファイルをディレクトリから削除します.
書式　　DEL {〈パス名〉|〈ファイル名〉}
別表記　ERASE {〈パス名〉|〈ファイル名〉}

解　説

〈ファイル名〉で指定されたファイルがディレクトリから削除されます. なおファイル名にはワイルドカード(*, ?)を使用することができます. 〈パス名〉のみ(ディレクトリ名のみ)を指定した場合, またはファイル名を〝*.*〟とした場合には, そのディレクトリに含まれるすべてのファイルが削除の対象となります.

サンプル・オペレーション

DEL コマンドを用いてファイルを削除してみます. カレント・ディレクトリは¥FUNC です.

```
A>dir ……ディレクトリの内容を表示する. カレント・ディレクトリは〝¥FUNC〟

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

.            <DIR>      84-02-22  22:04
..           <DIR>      84-02-22  22:04
HEXOUT   ASM       529  84-02-06  16:36
HEXOUT   OBJ       122  84-02-21  15:25
HEXOUT   LST      1791  84-02-20  22:54
FUNC-45  ASM       896  84-02-16  18:31
FUNC-45  COM       216  84-02-21  16:25
FUNC-45  LST      2727  84-02-21  16:25
FUNC-25  ASM       768  84-02-15  17:28
FUNC-25  COM       139  84-02-21  16:06
FUNC-25  LST      2186  84-02-21  16:06
AUXIN    ASM       185  84-02-19  21:13
      12 個のファイルがあります
  807936 バイトが使用可能です

A>del hexout.asm ……〝HEXOUT.ASM〟を削除する

A>erase *.lst ……拡張子が〝.LST〟のファイルをすべて削除する. ERASEはDELの別表記

A>dir

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

.            <DIR>      84-02-22  22:04
..           <DIR>      84-02-22  22:04 ……〝HEXOUT.ASM〟が削除されている
HEXOUT   OBJ       122  84-02-21  15:25 ┐
FUNC-45  ASM       896  84-02-16  18:31 │
FUNC-45  COM       216  84-02-21  16:25 │
FUNC-25  ASM       768  84-02-15  17:28 ├ 〝.LST〟はすべて削除されている
FUNC-25  COM       139  84-02-21  16:06 │
AUXIN    ASM       185  84-02-19  21:13 ┘
       8 個のファイルがあります
  817152 バイトが使用可能です
```

```
A>del *.*↵……カレント・ディレクトリにあるファイルをすべて削除する
よろしいですか <Y/N>? y↵……"DEL *.* " とした場合，およびパラメータにパス名を指定した場合，確認を求めてくる．
                       └── "y" を入力するとすべてのファイルを削除する
A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

.            <DIR>      84-02-22  22:04 ┐
..           <DIR>      84-02-22  22:04 ┘ すべてのファイルが削除されている
       2 個のファイルがあります
  823296 バイトが使用可能です

A>■
```

DIR

内部 1 .25

機能　ディレクトリの内容を表示します．

書式　DIR　[{<d : > | <パス名> | <ファイル名>}] [／P] [／W]

解　説

パラメータで指定されたディレクトリやファイルに関するディレクトリ情報(ファイル名，ディレクトリ名，ファイルの大きさ，最後に編集された日時)を表示します．

DIR コマンドで最も多く使われるのは，パラメータをすべて省略した形式です．次のように入力するとカレント・ドライブのカレント・ディレクトリに関する情報が表示されます．

A > DIR↵

ファイル名の指定にはワイルドカード(*，?)を使用することにより，必要なファイルに関するディレクトリ情報のみを表示することができます．またディレクトリのファイルが多くなったときや，ファイル名の一部を忘れてしまったときに，目的のファイルを捜し出す場合にもたいへん便利です．

次の 3 組の DIR コマンドはそれぞれ同じ働きをします．

DIR	DIR　*．*
DIR　filename	DIR　filename．*
DIR　．ext	DIR　*．ext

DIR コマンドには次の 2 つのスイッチがあります．

／P スイッチはページモードの指定を行うもので，画面がいっぱいになった時点で表示を停止します．続きを見るには，任意のキーを押します．

これと似た機能を DIR コマンドと MORE コマンドを次に示すようにパイプでつなぐことによっても実現できます．

```
A > DIR | MORE ↵
```

ただし，両者を実現しているアルゴリズムはまったく異なるものです．パイプを用いた場合，ディスク上にテンポラリファイルを開くので〝DIR／P〟と比べて実行速度が極端に落ちてしまいます．単にディレクトリの内容をページごとに表示するのであれば，〝DIR／P〟を使った方がよいでしょう．

／Wスイッチはワイド・ディスプレイの指定を行うもので，ファイル名またはディレクトリ名のみが1行に5つずつ表示されます．日付など他の情報は表示されません．

サンプル・オペレーション

まず，DIRだけを入力してカレント・ディレクトリ(ルート)の内容を表示した後，サブ・ディレクトリFUNCにカレント・ディレクトリを変更して，そこでワイルドカードを使った例を示します．次にサブ・ディレクトリCOMに移り，／Wおよび／Pスイッチを使った例を示します．

このときのディレクトリ構造はFig. 1.4のようになっています．

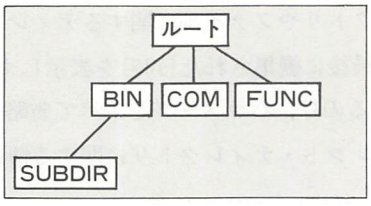


Fig. 1．4　ディレクトリの状態

```
A>dir ↵……カレント・ディレクトリ（ルート）の内容を表示する

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
BIN         <DIR>      84-02-22  20:32 ┐
COM         <DIR>      84-02-22  20:21 ┘ サブ・ディレクトリ
AUTOEXEC BAT       47  84-02-22  20:37
SORT     EXE     1360  83-09-04  23:54
FUNC        <DIR>      84-02-22  22:04 ……サブ・ディレクトリ
        7 個のファイルがあります
   805888 バイトが使用可能です

A>chdir func ↵……〝FUNC〟にカレント・ディレクトリを変更する

A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

.           <DIR>      84-02-22  22:04
..          <DIR>      84-02-22  22:04
HEXOUT   ASM      529  84-02-06  16:36 ┐
HEXOUT   OBJ      122  84-02-21  15:25 │
            1791  84-02-20  22:54 │ サブ・ディレクトリ〝FUNC〟にあるファイル
                  84-02-16  18:31 │
                            16:25 ┘
```

```
FUNC-25  COM       139
FUNC-25  LST      2186  84-02-
AUXIN    ASM       185  84-02-19   21:18
      12 個のファイルがあります
   805888 バイトが使用可能です
```

A>dir func*.??m ……"FUNC"ではじまりMで終わるファイルをすべてリストアップする(DIR FUNC＊.??Mでも同じ)

```
 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

FUNC-45  ASM       896  84-02-16  18:31
FUNC-45  COM       216  84-02-21  16:25
FUNC-25  ASM       768  84-02-15  17:28
FUNC-25  COM       139  84-02-21  16:06
       4 個のファイルがあります
   805888 バイトが使用可能です
```

A>dir ¥bin ……サブ・ディレクトリ "BIN" の内容を表示する（カレント・ディレクトリはFUNCであり，BINはルートのサブ・ディレクトリであるのでパスネームは "¥BIN" と指定する．"DIR BIN" とするとエラーになる．¥はルートを表す）

```
 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥bin

         <DIR>      84-02-22  20:32
         <DIR>      84-02-22  20:32
..
DEBUG    COM    12146  83-09-05   0:35
EDLIN    COM     9124  83-09-17  12:57
         EXE     1656  83-09-04  21:14
HELP     EXE    [illegible]  83-10-30  16:00
MASM     EXE    80256  [illegible]
DUMP     COM     1670  83-09-1[illegible]
SUBDIR       <DIR>      84-02-22  21:40
      11 個のファイルがあります
   805888 バイトが使用可能です
```

A>chdir ¥com ……サブ・ディレクトリ "COM" に移る

A>dir /w ……／Wスイッチを付けてディレクトリの内容を表示する．／W スイッチを付けると1行に5つずつ，ファイル名およびディレクトリ名のみが表示される

```
 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥com

[.]             [..]            CHKDSK   COM    MORE     COM    RECOVER  COM
SYS      COM    FC       EXE    FIND     EXE    SORT     EXE    ASSIGN   COM
COPY2    COM    COPYA    COM    DICM     COM    DISKCOPY COM    FORMAT   COM
KEY      COM    PRINT    COM    SPEED    COM    SWITCH   COM    USKCGM   COM
COPYA    DOC    README   DOC    CC       BAT    ASM      BAT
      24 個のファイルがあります
   805888 バイトが使用可能です
```

A>dir /p ……／Pスイッチを付けてディレクトリの内容を表示する．／Pスイッチを付けるとディレクトリの内容が1画面ごとに表示される

```
 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥com

.            <DIR>      84-02-22  20:21
..           <DIR>      84-02-22  20:21
CHKDSK   COM     6774  83-09-04  23:29
MORE     COM     4369  83-09-04  23:44
RECOVER  COM     2607  83-09-04  23:51
SYS      COM      968  83-09-04  23:57
FC       EXE     2642  83-09-04  23:40
FIND     EXE     6430  83-09-04  23:41
SORT     EXE     1360  83-09-04  23:54
ASSIGN   COM     1069  83-08-12  14:32
COPY2    COM     3312  83-08-27  11:18
COPYA    COM     1319  83-09-13  19:46
DICM     COM     7936  83-09-15  15:30
DISKCOPY COM     6224  83-08-25  13:52
FORMAT   COM    11878  83-10-29  15:46
KEY      COM     4531  83-10-30  19:46
PRINT    COM     4697  83-09-18  18:02
SPEED    COM     1209  83-08-10  15:07
SWITCH   COM     2054  83-10-28  16:30
USKCGM   COM     4130  83-10-30  19:43
COPYA    DOC     1831  83-09-19  15:07
README   DOC     2754  83-10-28  17:22
CC       BAT       65  84-02-22   3:31
```

1画面分

```
準備ができたらどれかキーを押してください ……ここで何かキーを押すと次が表示される
ASM      BAT        66  84-02-21   15:28
       24 個のファイルがあります
   805888 バイトが使用可能です

A>■
```

DISKCOPY

外部1.25

機能　ディスク単位でのバックアップコピー，および照合を行います．

書式　DISKCOPY　[〈d1：〉][〈d2：〉][／V]

解　説

第1パラメータ〈d1：〉で指定されたドライブのディスクの内容を，すべて第2パラメータ〈d2：〉で指定されたドライブにコピーします．

操作ミスや事故などによってディスクの内容の一部または全部が壊されてしまうことがあるため，このような事故に備えて，こまめにディスクのバックアップを取るようにしなければなりません．

DISKCOPYコマンドは，COPYコマンドと違ってディスクにあるすべてのファイル，ディレクトリなどを物理的に他のディスクへコピーしますから，DISKCOPYコマンドを実行すると受け側のディスクの内容はすべて失われてしまいます．また異なる種類のディスクの間でコピーはできません．

第2パラメータが省略された場合や両方のパラメータが省略された場合には，それぞれのドライブ指定はカレント・ドライブになります．

2ドライブ以上のシステムでは

A > DISKCOPY A：　B：↵

と指定するのが一般的です．この場合は，ドライブAのディスクの内容がドライブBのディスクにそのままコピーされます．

1ドライブのシステムでは，

A > DISKCOPY A：　A：↵

または，

A > DISKCOPY ↵

と指定しますが，この場合はメッセージに従って2枚のディスケットを交換しながらコピーを行います．

／Vスイッチを指定すると，ディスクのコピーではなく2つのディスクの照合のみを行います．このスイッチは，DISKCOPYコマンドを使ってコピーしたディスクが間違なくコピーされているかを確認するためのものです．重要なファイルのはいっているディスクのバックアップを行った場合には，〝DISKCOPY［〈d1：〉］［〈d2：〉］／V〞を指定して照合した方がよいでしょう．（参考：COPY）

サンプル・オペレーション

2ドライブ・システムの例では，DISKCOPYコマンドを使ってドライブAのディスクの内容をドライブBのディスクにコピーを行った後，〝DISKCOPY A：B：／V〞と入力して，コピーが完全に行われているかを確認しています．

1ドライブでのディスクのバックアップの例では，メッセージに従って2枚のディスケットを交換しながらコピーを行っています．

```
A>diskcopy a: b:↵……ドライブAからBへディスクをバックアップする

DISKCOPY version 2.0

ディスクのコピーを行います

送り側ディスクをドライブ A: に挿入してください
受け側ディスクをドライブ B: に挿入してください
準備ができたらどれかのキーを押してください ……送り側をドライブA，受け側をドライブBに入れて何かキーを押す

コピーは終了しました
別のディスクをコピーしますか <Y/N>? n↵……別のディスクをコピーするときは〝y〞を入力する
A>diskcopy a: b: /v ↵……ドライブAのディスクとドライブBのディスクの照合をする

DISKCOPY version 2.0

ディスクの照合を行います

送り側ディスクをドライブ A: に挿入してください
受け側ディスクをドライブ B: に挿入してください
準備ができたらどれかのキーを押してください ……何かキーを押すと照合がはじまる

照合を終了しました
別のディスクを照合しますか <Y/N>? n↵
A>diskcopy ↵……ドライブ名を省略してDISKCOPYのみを入力すると，カレント・ドライブで2枚のディスケットを交互に抜き差ししながらディスクのバックアップを行う（1ドライブ用のディスクのバックアップ，2ドライブ以上のシステムの場合，
                                普通は〝DISKCOPY A：B：〞などの方を用いる）
DISKCOPY version 2.0

ディスクのコピーを行います

受け側ディスクをドライブ A: に挿入してください
準備ができたらどれかのキーを押してください
送り側ディスクをドライブ A: に挿入してください
準備ができたらどれかのキーを押してください
受け側ディスクをドライブ A: に挿入してください
準備ができたらどれかのキーを押してください ↵……何かキーを押す

コピーは終了しました
別のディスクをコピーしますか <Y/N>? n↵
A>■
```

EXIT

内部２．０

機能　現在の〝COMMAND．COM〟を呼び出したプログラムがあればそのプログラムに戻ります．
書式　EXIT

解　説

EXITコマンドは，アプリケーション・プログラムや〝COMMAND．COM〟自身が〝COMMAND．COM〟を呼び出したときに，その呼び出したプログラムに戻る場合に使用するコマンドです．

アプリケーション・プログラム(エディタやBASICなど)を使用中に，〝COMMAND．COM〟の機能を利用する場合(たとえば，ディレクトリの内容を表示する，あるいはファイルをプリントアウトするといった場合)，実行中のアプリケーション・プログラムを中止して〝COMMAND．COM〟へ戻り，目的とする仕事をしてから再びそのアプリケーション・プログラムを起動しなければなりません．この作業は一般にたいへんわずらわしいものです．

ところが，もしそのアプリケーション・プログラムに〝COMMAND．COM〟を呼び出す機能があれば，〝COMMAND．COM〟を呼び出した後目的の仕事を行い，そのアプリケーション・プログラムに戻れば〝COMMAND．COM〟を呼び出した直前の状態に戻ることができるので，それまでしていたことをすぐに再開できます．この〝COMMAND．COM〟からアプリケーション・プログラムへ戻るためのコマンドがEXITコマンドなのです．

ただし，そのアプリケーション・プログラムに〝COMMAND．COM〟を呼び出す機能がない場合には，EXITコマンドはまったくその意味をなしません．たとえば，EDLINにはそういった機能はありませんから，ディレクトリの内容を表示するときなどは，一度EDLINを終了して，目的とする仕事を終えてからEDLINを再起動しなければなりません．つまり，EXITコマンドは〝COMMAND．COM〟を呼び出す機能を持ったプログラムとともに使うことで，初めてその機能を果たすのです．

サンプル・オペレーション

〝COMMAND．COM〟自身も〝COMMAND．COM〟を呼び出す機能を持ったプログラムの１つです．

```
A > COMMAND ↵
```

と入力すると，ごく普通の外部コマンドのように，〝COMMAND．COM〟が起動されます．このこと自体は単に〝COMMAND．COM〟の分だけメモリが減るだけで，なんら意味を持ちません

が，これをリダイレクトとともに用いてEXITコマンドの働きを見ていきます．

まず，DIRなどのコマンドをバッチファイルのように収めたファイル〝OPR〟を作っておきます．その最後にはかならずEXITコマンドを加えます(そうしないと，実行が終わっても何も入力できなくなる)．そして，このファイルを入力に，新しいファイル〝OPR．TXT〟を出力にして〝COMMAND．COM〟を起動すると，〝OPR〟の中に収めておいたコマンドがキーボードから入力されたかのように実行され，その結果が〝OPR．TXT〟に収められます．具体的には，次のように入力します．

A > COMMAND < OPR > OPR．TXT ↵

なお，実行中はファイル〝OPR．TXT〟に出力していますので，スクリーンには何も現れません．

Fig.1.5は，このオペレーション中に起動されるプログラムと，その中で実行されているコマンドを表したものです．

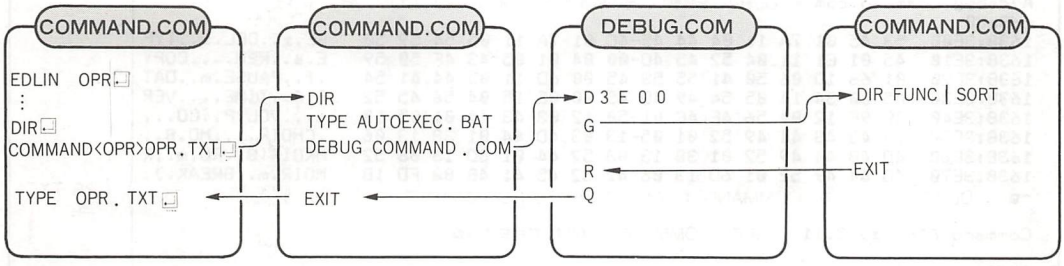


Fig. 1．5　起動されるプロセスと実行されるコマンド

```
A>edlin opr ↵……EDLINでファイル名〝OPR〟のファイルを作る
新しいファイルです
*i ↵……インサートコマンド
     1:*dir ↵……カレント・ディレクトリを表示する
     2:*type autoexec.bat ↵……〝AUTOEXEC．BAT〟をタイプアウトする
     3:*debug command.com ↵……〝DEBUG．COM〟を起動して〝COMMAND．COM〟をロードする
     4:*d3e00 ↵……オフセット3E00Hからダンプする
     5:*g ↵……ロードしたプログラム〝COMMAND．COM〟を実行する
     6:*dir func | sort ↵……ディレクトリ〝FUNC〟をソートし，出力する
     7:*exit ↵……〝DEBUG．COM〟へ戻る
     8:*r ↵……レジスタ表示
     9:*q ↵……〝DEBUG．COM〟を抜ける
    10:*exit ↵……もとの〝COMMAND．COM〟へ戻る
    11:*^Z ↵……CTRL-Z↵でインサートモードを抜ける
*e ↵

A>dir ↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
KNJDIC   SYS     56320  83-09-20  14:22
BIN          <DIR>      84-02-22  20:32
COM          <DIR>      84-02-22  20:21
AUTOEXEC BAT        46  84-02-23  17:48
```

```
SORT      EXE      1360  83-09-04  23:54
FUNC           <DIR>      84-02-22  22:04
OPR                89  84-02-23  17:52 ……今作ったファイル
        8 個のファイルがあります
   804864 バイトが使用可能です

A>command < opr > opr.txt↵ ┐入力をファイル "OPR",出力を "OPR . TXT" にしてCOMMANDを起動する．すると "OPR"の内容があたかもキーボードから入力されたように実行され，その結果が "OPR . TXT"に落ちる．スクリーンには何も表示されない．バッチ処理でも同じことができるが，その結果をファイルにリダイレクトできない．
A>type opr.txt↵ ……"OPR . TXT" をタイプする ┘
Command ﾊﾞｰｼﾞｮﾝ 2.11 ……子の"COMMAND . COM" が起動する

A>dir

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19
                56320  83-09-20  14:22
                       84-02-22  20:32
COM            <DIR>             09:21
OPR                89  84-02-2
OPR      TXT        0  84-02-23  17:5
        9 個のファイルがあります
   803840 バイトが使用可能です

A>type autoexec.bat
date
time
echo off
path a:¥;a:¥com;a:¥bin

A>debug command.com ……DEBUGを起動し，その管理下に "COMMAND . COM" をロードする
-d3e00
1638:3E00  53 45 01 7A 11 04 44 45-4C 01 7A 11 05 54 59 50   SE.z..DEL.z..TYP
1638:3E10  45 01 E1 11 04 52 45 4D-00 04 01 05 43 4F 50 59   E.a..REM....COPY
1638:3E20  01 66 1D 06 50 41 55 53-45 00 6D 11 05 44 41 54   .f..PAUSE.m..DAT
1638:3E30  45 00 34 14 05 54 49 4D-45 00 0E 15 04 56 45 52   E.4..TIME....VER
1638:3E40  00 9F 12 04 56 4F 4C 01-50 12 03 43 44 01 05 13   ....VOL.P..CD...
1638:3E50  06 43 48 44 49 52 01 05-13 03 4D 44 01 38 13 06   .CHDIR....MD.8..
1638:3E60  4D 4B 44 49 52 01 38 13-03 52 44 01 6D 13 06 52   MKDIR.8..RD.m..R
1638:3E70  4D 44 49 52 01 6D 13 06-42 52 45 41 4B 00 FD 1B   MDIR.m..BREAK.}.
-g ……DEBUGがロードした "COMMAND . COM" のエントリ・アドレスにジャンプ

Command ﾊﾞｰｼﾞｮﾝ 2.11 …… 孫の "COMMAND . COM" が起動される

A>dir func | sort


       12 個のファイルがあります
    801792 バイトが使用可能です
 ディレクトリは A:¥func
 ドライブ A: のボリュームラベルは MS-DOS HB
.              <DIR>      84-02-22  22:04
..             <DIR>      84-02-22  22:04
AUXIN     ASM        185  84-02-19  21:13
FUNC-25   ASM        768  84-02-15  17:28
FUNC-25   COM        139  84-02-21  16:06
FUNC-25   LST       2186  84-02-21  16:06
FUNC-45   ASM        896  84-02-16  18:31
FUNC-45   COM        216  84-02-21  16:25
FUNC-45   LST       2727  84-02-21  16:25
HEXOUT    ASM        529  84-02-06  16:36
HEXOUT    LST       1791  84-02-20  22:54
HEXOUT    OBJ        122  84-02-21  15:25

A>exit …… 孫の "COMMAND . COM" から抜け，DEBUGに戻る

Program terminated normally ……プログラムが正常に終了したというDEBUGからのメッセージ
-r
AX=0000  BX=0000  CX=4326  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1638  ES=1638  SS=1638  CS=1638  IP=0100   NV UP DI PL NZ NA PO NC
1638:0100 E9CD0B        JMP     0CD0
-q

A>exit ……子の "COMMAND . COM" から抜け，もとの "COMMAND . COM" に戻る．このEXITコマンドで
         COMMAND<OPR>OPR . TXTは終了する
A>■
```

"OPR.TXT" の内容

FIND

外部2.0

機能　1つまたは複数のファイルから，任意の文字列を検索します．
書式　FIND ［／V］［／C］［／N］〝〈文字列〉〞［〈ファイル名〉……］

解　説

FINDコマンドはSORT，MOREと同様フィルタの一種です．指定したファイルから指定した文字列を検索し，それを含む行を表示します．プログラム開発には，文字列の検索機能はなくてはならないものですし，他のコマンドと併せてバッチファイルを作成する場合にも有効に利用できます．

FINDコマンドには次の3つのスイッチがあります．

／V　指定した文字列を含まない行を返す
／C　指定した文字列を含む行数のみを返す
／N　FINDの結果に，その行が初めから何行目にあるかという行番号をその行の先頭に付けて返す

これらのスイッチは1つだけではなく，2つまとめて指定することができます．／Vと／Cを同時に指定すると，FINDコマンドは指定した文字列を含まない行数を返し，／Vと／Nを同時に指定すると，その文字列を含まない行に行番号を付けて返します．

しかし，／Cと／Nの組合せは無意味ですから，この場合／Nの指定は無視され，／Cの指定のみが有効となり，FINDコマンドは指定した文字列を含む行数を返します．同様に／V，／C，／Nのすべてのスイッチの指定も無意味なので，FINDコマンドは／Vと／Cを指定した場合と同じ結果を返します．

ファイル名を省略すると，標準入力(キーボード)がFINDコマンドの入力となります．

またFINDコマンドはパイプ機能と組み合せることもできます．たとえば

```
A > DIR | FIND "DIR" ↵
```

などと入力します．

サンプル・オペレーション

初めの例は，ファイル名を指定する形で，Cのソース・プログラムからいくつかの文字列を検索しており，次は，パイプ処理を用いてディレクトリの内容から文字列を検索しています．

```
A>type sort.c↵……"SORT．C" をタイプアウトする
#include <stdio.h>

main()
{
        int     a[10];
        int     i,j,tmp;

        for (i=0;i<10;i++){
                printf("please input integer  ? ");
                scanf("%d¥r¥n",&a[i]);
        }

        for (i=0;i<9;i++){
                for (j=i+1;j<10;j++){
                        if (a[i]>a[j]){
                                tmp=a[i];
                                a[i]=a[j];
                                a[j]=tmp;
                        }
                }
        }

        for (i=0;i<10;i++) printf("%d ",a[i]);
}

A>find "for" sort.c↵……"SORT．C" の中から "for" を含む行を捜す

---------- sort.c
        for (i=0;i<10;i++){
        for (i=0;i<9;i++){
                for (j=i+1;j<10;j++){
        for (i=0;i<10;i++) printf("%d ",a[i]);

A>find /v ";" sort.c↵……"SORT．C" の中で "；" を含まない行を捜す

---------- sort.c
#include <stdio.h>

main()
{

        }

                        if (a[i]>a[j]){
                        }
                }
        }

}
A>find /c "[" sort.c↵……"SORT．C" の中で "［" を含む行の行数を求める

---------- sort.c: 7

A>find /n "for" sort.c↵……"SORT．C" の中で "for" を含む行を捜し，それが何行目かを表示する

---------- sort.c
[8]     for (i=0;i<10;i++){
[13]    for (i=0;i<9;i++){
[14]            for (j=i+1;j<10;j++){
[23]    for (i=0;i<10;i++) printf("%d ",a[i]);

A>chdir ¥↵

A>dir↵

 ドライブ A: のボリュームラベルは PC-9801
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
BIN          <DIR>      84-02-25  14:05
COM          <DIR>      84-02-25  14:05
                                        ……"ＤＩＲ" を含む行．すなわちサブ・ディレクトリ
HC           <DIR>      84-02-28  18:24
C            <DIR>      84-03-06   7:27
       14 個のファイルがあります
   784384 バイトが使用可能です
```

```
A>dir ¦ find /v/n "DIR"↵……パイプを用いてディレクトリの内容のうちで "DIR" を含まない行を捜し，何行目かを表示する
[1]
[2] ドライブ A: のボリュームラベルは PC-9801
[3] ディレクトリは A:¥
[4]
[5]COMMAND  COM     17190  83-09-19  16:19
[6]KNJDIC   SYS     56320  83-09-20  14:22
[9]AUTOEXEC BAT        46  84-02-23  17:48
[10]LOGIN    BAT       374  84-02-25  14:54
[12]CHK      BAT       152  84-02-25  15:25
[13]KILL     BAT        42  84-02-25   8:47
[14]FUNC-17  ASM       512  84-02-20  22:02
[15]DUMP     ASM      1479  84-02-20  22:58
[16]MEMDUMP  ASM      1152  84-02-27  22:00
[19]%PIPE1   $$$         0  84-03-06   8:07 }
[20]%PIPE2   $$$         0  84-03-06   8:07 } パイプ処理のためのテンポラリファイル．処理が終わると自動的に削除される
[21]        16 個のファイルがあります
[22]    783360 バイトが使用可能です

A>■
```

FORMAT

外部 1 .25

機能　ディスクを MS-DOS で使えるようにフォーマット(初期化)します．

書式　FORMAT [<d : >] [／S] [／V]

解　説

FORMAT コマンドは，MS-DOS が読み書きできるようにディスクを初期化します．FORMAT コマンドで初期化を行っていないディスクは，読み書きができません．

FORMAT コマンドには次の 2 つのスイッチがあります．

／S　ディスクをフォーマットした後に，システムファイル(〝IO . SYS〟，〝MSDOS . SYS〟，〝COMMAND . COM〟)をフォーマットされたディスクへ書き込みます．以後，このディスクを使って MS-DOS を起動することができるようになります．

／V　ディスクにボリュームラベルを付けます．フォーマット終了後にボリュームラベルの入力待ちになりますから，キーボードから英文字で 11 文字，漢字で 5 文字以内で入力します．ボリュームラベルは一度付けると再びフォーマットするまで，〝COMMAND . COM〟の機能からは変更することができません．

FORMAT コマンドは〝IO . SYS〟同様，機種別に各メーカーで作られるため，機種によっては，FORMAT コマンドの持つスイッチは上に示したものと異なっていたり，あるいはそれ以外のスイッチを持っている場合があります．詳しくは使用する機種のマニュアルを参照してください．

なお，それまで MS-DOS あるいは他の OS, DISK BASIC などで使っていたディスクを FORMAT コマンドでフォーマットしてしまうと，ディスク中のデータはすべて失われてしまいます．FORMAT コマンドを使用する際には十分注意してください．

サンプル・オペレーション

このオペレーションはNEC PC-9801で行われています．

まずフロッピーディスクのフォーマットの例ですが，ドライブBのディスクを／Vと／Sスイッチを付けてフォーマットします．通常2ドライブ・システムでディスクをフォーマットする場合，システムディスク(FORMATコマンドのはいっている)をドライブAに入れ，フォーマットしたいディスクをドライブBに入れます．

そこで次のように入力すると，ドライブBのディスクがフォーマットされます．

A > FORMAT B : ↵

1ドライブ・システムの場合，FORMATコマンドを起動した後に，フォーマットしたいディスクと入れ換えることになります．

次にハードディスクのフォーマットの例を示します．PC-9801用のMS-DOSのFORMATコマンドには／Hスイッチが用意されており，これを用いてハードディスクをフォーマットすることができます．

```
A>format b: /v/s ……ドライブBのディスクをフォーマットする．／Vおよび／Sスイッチはそれぞれボリュームラベルを付ける，システムを転送するという指定
Format  Version 2.2

新しいディスクをドライブ B: に挿入し
どれかのキーを押してください ……ドライブBにフォーマットしたいディスケットを入れ何かキーを押すとフォーマットがはじまる

目的のディスクは８" ２Ｄです ……フォーマットしようとするディスクの種類を自動的に判定する（この場合8インチ両面倍密）
フォーマット中です．．．フォーマットが終了しました
システムを転送しました……／Sスイッチを指定するとシステム "ＩＯ．SYS","MSDOS．SYS","COMMAND．COM"をフォーマットが終了したディスクに転送する
ディスクのボリュームラベルを入力してください……／Vスイッチを指定するとボリュームラベルの入力待ちになる
漢字 － ５文字、英数字 － １１文字、無名称 － リターンキー  ms-dos hb↵
                                                  └ボリュームラベルを入力する
  1250304 バイト：全ディスク空間
    76800 バイト：システム空間
  1173504 バイト：使用可能ディスク空間

別のディスクをフォーマットしますか <Y/N> ?n ……ここで"y↵"とすると続いてフォーマットができる
A>dir b: ……今フォーマットしたディスクのディレクトリの内容を見る

 ドライブ B: のボリュームラベルは MS-DOS HB
 ディレクトリは B:¥

COMMAND  COM    17190  83-09-19  16:19 }
       1 個のファイルがあります       } "COMMAND．COM" が転送されている．"ＩＯ．SYS" と "MSDOS SYS" も転送されているが，この2つはファイルとして見ることはできない
  1173504 バイトが使用可能です       }

A>format /h ……ハードディスクをフォーマットする

Format  Version 2.2


固定ディスクの装置番号を入力してください [1,2] = 1 ……2台までつなげるハードディスクのうちどちらをフォーマットするかを決める．この場合ハードディスクは1台のみなので1を入力する
モードを選択してください
1:マップ 2:領域確保 3:領域解放 4:装置初期化 5:装置変更 6:終了 = 1 ……マップを見る

        現在の装置使用状況          装置番号 1 }
        ＭＳ－ＤＯＳ                領域無し   }
        他のＯＳ                    領域無し   } 5Mバイトすべて空き
        空き領域                     5 MB      }
```

```
モードを選択してください
1:マップ 2:領域確保 3:領域解放 4:装置初期化 5:装置変更 6:終了 = 4↵……ハードディスク全体をフォーマットする

        現在の装置使用状況          装置番号 1
        ＭＳ－ＤＯＳ                領域無し
        他のＯＳ                    領域無し
        空き領域                     5 MB

装置全体を初期化します、いいですか <Y/N> ? y↵
装置を初期化中です
  残り [0] メガバイト
       └── ハードディスクの全容量(この場合5)から1Mバイトフォーマットが終わるたびにカウントダウンされる
モードを選択してください
1:マップ 2:領域確保 3:領域解放 4:装置初期化 5:装置変更 6:終了 = 2↵……MS-DOS用にハードディスクの領域を確保する

        現在の装置使用状況          装置番号 1
        ＭＳ－ＤＯＳ                領域無し
        他のＯＳ                    領域無し
        空き領域                     5 MB

領域の大きさを入力してください [1- 5 MB] = 4↵……MS-DOS用に確保したい領域の大きさを1Mバイト単位で入力する

領域確保中です
  残り [0] メガバイト
       └── 上で入力した値から1Mバイトずつ領域確保が終わるたびにカウントダウンされる
ディスクのボリュームラベルを入力してください
漢字 － ５文字、英数字 － １１文字、無名称 － リターンキー  hd_system↵……ボリュームラベルを入力する

モードを選択してください
1:マップ 2:領域確保 3:領域解放 4:装置初期化 5:装置変更 6:終了 = 6↵……FORMATを終了しCOMMANDに戻る
A>■
```

MKDIR(MD)

内部２．０

機能　新しいサブ・ディレクトリを作成します.

書式　MKDIR 〈パス名〉

別表記　MD 〈パス名〉

解　説

ディスクをFORMATコマンドでフォーマットした状態では，ルート・ディレクトリが存在するだけなので，他のサブ・ディレクトリはユーザーの目的に応じた階層的ディレクトリ構造をMKDIRコマンドによって構築します.

MKDIRコマンドを用いてサブ・ディレクトリを作ると，その下に今作ったサブ・ディレクトリ自身を表す〝.〟ディレクトリと，その親ディレクトリを表す〝..〟ディレクトリが自動的に作成されます.

ディレクトリ名(パス名のいちばん最後)の付け方は，ファイル名の場合と同じです．ただし，ファイル名とディレクトリ名は名前では区別されませんので，すでに存在するファイルと同じ名前のディレクトリを作ることはできません．またパス名は，作ろうとするディレクトリ名(パス名の最後)以外のディレクトリはすでに存在していなければなりません．たとえば，ルート・ディレクトリしかないときに次のような入力を行って，２階層のディレクトリを一度に作ることはできません.

```
A > MKDIR USR¥BIN↵
```

この場合は，ディレクトリ USR を作り，その後に BIN を作るといった手順を踏まなければなりません．

(参考：CHDIR，RMDIR)

サンプル・オペレーション

MKDIR コマンドで実際にサブ・ディレクトリを作ってみます．最初にディレクトリ構造は Fig. 1.6 のようになっているものとします．

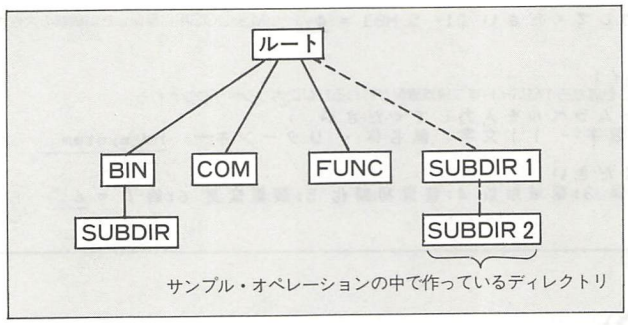


Fig. 1.6　ディレクトリの状態

```
A>mkdir subdir1↵……ルートの下にサブ・ディレクトリ "SUBDIR1" を作る(カレント・ディレクトリはルート)

A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM     17190   83-09-19   16:19
                 47020   83-09-20   14:22
FUNC             <DIR>       -22    22:04
OPR                 89   84-
OPR      TXT       2218   84-02-23   1
SUBDIR1          <DIR>      84-02-23    1:19 ……今作ったサブ・ディレクトリ
       10 個のファイルがあります
   800768 バイトが使用可能です

A>dir subdir1↵……サブ・ディレクトリ "SUBDIR1" の内容を表示する

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥subdir1

.              <DIR>      84-02-23    1:19 ｝ディレクトリを作ると自動的に作られる
..             <DIR>      84-02-23    1:19 ｝"." はカレント・ディレクトリ，".." はその親ディレクトリ
        2 個のファイルがあります
   800768 バイトが使用可能です

A>md subdir1¥subdir2↵……"SUBDIR1" の下に "SUBDIR2" を作る

A>chdir subdir1¥subdir2↵……"SUBDIR2" に移る

A>dir↵……カレント・ディレクトリ "SUBDIR2" の内容を表示する

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥subdir1¥subdir2

.              <DIR>      84-02-23    1:20
..             <DIR>      84-02-23    1:20
        2 個のファイルがあります
   799744 バイトが使用可能です
```

```
A>chdir ..↵……親ディレクトリに移る
A>dir↵……ディレクトリの内容を表示する
 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥subdir1

.            <DIR>      84-02-23   1:19
..           <DIR>      84-02-23   1:19
SUBDIR2      <DIR>      84-02-23   1:20
       3 個のファイルがあります
  799744 バイトが使用可能です

A>chdir ¥↵……ルートに戻る．この場合親ディレクトリがルートなので，"CHDIR . ." でも同様にルートに戻ることができる
A>■
```

MORE

外部2.0

機能　標準入力から読み込んだ内容をスクリーンに1画面ずつ出力します．

書式　MORE

解　説

MOREコマンドは，フィルタの一種で，標準入力から読み込んだ内容を一度に1画面(23行)ずつ，標準出力(通常スクリーン)へ出力するものです．

出力が1画面を超えるような大きなファイルをタイプアウトするときなど，スクリーンが次々とスクロールしてしまい，内容を見るにはCTRL-Sで出力を一時止めなければなりません．しかし，MOREコマンドを使えば，そういったわずらわしさを解消することができます．

MOREコマンドは，画面がいっぱいになると，画面の最下行に

```
--More--
```

と表示して，いったん出力を止めます．ここで任意のキーを押すと，次の1画面分の表示を行い，これをMOREへの入力データが終わるまで繰り返します．

MOREコマンドで注意しなければならないのは，標準入力以外からの入力が許されないということです．たとえば，MOREコマンドでファイル "FILE．TXT" を出力する場合，

```
A > MORE  FILE．TXT↵
```

としても "FILE．TXT" は出力されません．この場合は "FILE．TXT" を標準入力にして

```
A > MORE < FILE．TXT↵
```

あるいは

```
A > TYPE FILE . TXT | MORE ↵
```

としなければなりません．

(参考：FIND，SORT)

サンプル・オペレーション

まず，サブ・ディレクトリCOMの内容をパイプ処理を使って1画面ずつ表示してみます．次にアセンブラのソースファイル"HEXOUT . ASM"を，MOREコマンドを使って1画面ずつ表示する例を示します．前の例ではパイプ処理を使っていましたが，この場合，入力のリダイレクトによって実現しています．

```
A>dir com ¦ more ↵……サブ・ディレクトリ "COM" の内容をMOREに送り1画面ずつ表示させる

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥com

.            <DIR>      84-02-22  20:21
..           <DIR>      84-02-22  20:21
CHKDSK   COM     6774  83-09-04  23:29
MORE     COM     4369  83-09-04  23:44
RECOVER  COM     2607  83-09-04  23:51
SYS      COM      968  83-09-04  23:57
FC       EXE     2642  83-09-04  23:40
FIND     EXE     6430  83-09-04  23:41
SORT     EXE     1360  83-09-04  23:54
ASSIGN   COM     1069  83-08-12  14:32
COPY2    COM     3312  83-08-27  11:18
COPYA    COM     1319  83-09-13  19:46
DICM     COM     7936  83-09-15  15:30
DISKCOPY COM     6224  83-08-25  13:52
FORMAT   COM    11878  83-10-29  15:46
KEY      COM     4531  83-10-30  19:46
PRINT    COM     4697  83-09-18  18:02
SPEED    COM     1209  83-08-10  15:07
SWITCH   COM     2054  83-10-28  16:30
-- More -- ↵……何かキーを押すと次が見られる
USKCGM   COM     4130  83-10-30  19:43
COPYA    DOC     1831  83-09-19  15:07
README   DOC     2754  83-10-28  17:22
CC       BAT       65  84-02-22   3:31
ASM      BAT       66  84-02-21  15:28
      24 個のファイルがあります
   797696 バイトが使用可能です

A>more < func¥hexout.asm ↵……MOREコマンドに "HEXOUT . ASM" の内容を送り込む．すると "HEXOUT . ASM" の内容
                                 が1画面ずつ表示される(MS-DOSのMOREは"MORE〈ファイル名〉"とは書けない．
;-------------------------------------; 必ず"MORE <〈ファイル名〉"もしくは"TYPE〈ファイル名〉¦ MORE"と
;       Hex out in AX                 ;                                          する)
;-------------------------------------;
;
cseg    segment
        assume  cs:cseg
        add     al,3ah
        cmp     al,3ah
-- More --
        jb      hexout0
        add     al,7
cseg    ends
;
        end

A>■
```

PATH

内部２．０

機能　外部コマンドを検索するディレクトリを設定／表示します．
書式　PATH　[〈パス名〉[；〈パス名〉] ……]

解　説

外部コマンドがカレント・ディレクトリにない場合，そのコマンドを検索するディレクトリの設定，あるいは現在の設定状況を表示するためのコマンドです．

通常〝COMMAND．COM〟は，カレント・ディレクトリからのみ外部コマンドを検索するようになっていますから，外部コマンドの置かれていないサブ・ディレクトリで外部コマンドを実行する場合には，PATH コマンドで外部コマンドのパスを設定しなければなりません．

PATH コマンドによってパスを設定すると，〈パス名〉で指定されたディレクトリからそのコマンドを検索し，実行することができます．またパス名を〝；〟で区切っていくつものパスを設定すると，指定したパスの順にコマンドを検索するようになります．

現在設定されているパスを確認するためには次のように入力します．

```
A > PATH↵
```

また，次のように入力すると現在設定されているパスが解除されます．

```
A > PATH ;↵
```

通常外部コマンド類を１つまたはいくつかのサブ・ディレクトリに入れておきますから，〝AUTOEXEC．BAT〟の中で PATH コマンドを使って，ブート時にパスを自動設定するようにすると，そのたびにパスを設定し直さないで済むのでたいへん便利です．

なお，PATH コマンドによってパスを設定すると，

```
PATH =[〈パス名〉……]
```

の形で MS-DOS の環境に登録され，MS-DOS は環境の中の PATH を外部コマンドを捜すパスとして用います．すなわち，SET コマンドを用いて

```
SET　PATH =[〈パス名〉[；〈パス名〉] ……]↵
```

としても同様です．また，バッチファイル中で〝%PATH%〟とすると設定されているパスを参照することができます．

（参考：SET，PROMPT）

サンプル・オペレーション

新しくディレクトリCALを作成し，その下でEDLINを使って"FACT．PAS"を作ってみます．最初サブ・ディレクトリCALの中でEDLINを呼び出そうとすると，エラーが出力されますが，これはCALの中に"EDLIN．COM"が置かれておらず，パスも設定していないからです．外部コマンド類はルート，サブ・ディレクトリBINとCOMの中に入れてあるので，パスを

A > PATH A : ¥ ; A : ¥COM ; A : ¥BIN ↵

と設定して，サブ・ディレクトリCALからも外部コマンドが使えるようにしています．

```
A>mkdir cal ……サブ・ディレクトリ "CAL" をカレント・ディレクトリ(ルート)に作る

A>chdir cal ……"CAL"に移る

A>dir

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥cal

.            <DIR>      84-02-23   1:40
..           <DIR>      84-02-23   1:40
       2 個のファイルがあります
  798720 バイトが使用可能です

A>edlin fact.pas ……EDLINで "FACT．PAS" を作ろうとしたが，カレント・ディレクトリに"EDLIN．COM"
コマンドまたはファイル名が違います   がないので，エラーになった

A>path ……パスの設定状況を表示する
パスは定義されていません
                ┌ドライブAの "COM"
A>path a:¥;a:¥com;a:¥bin ……パスを設定する   これ以後コマンド(".COM" ".EXE" ".BAT")がカレント・ディレクトリに
        └ドライブAのルート └ドライブAの"BIN"   ない場合には，ルート，"COM"，"BIN" の順でコマンドを捜すようになる
A>path
PATH=A:¥;A:¥COM;A:¥BIN ……パスが設定された

A>edlin fact.pas ……EDLINで "FACT．PAS" を作る．パスが設定してあるのでEDLINは起動する
新しいファイルです
*i ……インサートコマンド
     1:*program factorial(input,output);
     2:*
     3:*  var n : integer;
     4:*
     5:*  function fact(n : integer) : integer;
     6:*    begin
     7:*      if n=0 then fact:=1
     8:*             else fact:=n*fact(n-1);
     9:*    end;
    10:*
    11:*begin
    12:*  writeln;
    13:*  write('n = ? ');
    14:*  readln(n);
    15:*  writeln(n:5,' != ',fact(n));
    16:*end.
    17:*^Z
*e ……EDLINを抜け "COMMAND．COM" に戻る

A>■
```

PRINT

外部2.0

機能　他のコマンドの実行と並行して，テキストファイルなどをプリントアウトします．
書式　PRINT　[[〈ファイル名〉] [／T] [／C] [／P]] ……

解　説

PRINT コマンドは，他のコマンドを実行中に(バックグラウンドで)ファイルをプリントアウトするためのコマンドです．PRINT コマンドを使うと，ユーザーはプリントアウトと同時に他の仕事ができるため，オペレーションが能率的に行えます．

PRINT に続いて指定したファイル名はプリント待ち行列に登録され，順次プリンタに出力されます．プリント待ち行列には最大 10 個のファイル名が登録でき，その範囲内で PRINT に続くファイル名をいくつでも指定することができます．なおファイル名の指定にはワイルドカード(＊，？)を使うこともできますが，パス名を指定することはできません．

また，ファイル名を指定せずに PRINT のみを入力すると，現在のプリント待ち行列の内容が表示されます．

PRINT コマンドには次の 3 つのスイッチがあります．

／T　プリント待ち行列にあるすべてのファイル名を取り消し，プリント処理はその場で中止されます．プリンタには取消しのメッセージが出力され，用紙の改ページを行います．

／C　／C の直前のファイル名から／P スイッチの付いたファイル名の前までのすべてのファイル名をプリント待ち行列から削除します．

／P　／P の直前のファイル名から／C スイッチの付いたファイル名の前までのすべてのファイル名をプリント待ち行列に登録します．

／C および／P スイッチは，その前のファイル名から他のスイッチが現れるか，あるいは PRINT 文の終わりのファイル名まで有効ですが，次の PRINT コマンドまでは影響を及ぼしません．またスイッチを省略した場合／P が指定されたものとみなされます．

／C スイッチは，誤まって指定してしまったファイル名をプリント待ち行列から削除するためのスイッチですが，プリンタがデータ用のバッファを持っている場合には，／C スイッチによってプリントを取り消そうとしても，すでにすべてのファイルの内容がプリンタ内のバッファに登録されていることがあるため，プリントアウトの取消しはできない場合があります．

PRINT コマンドは，プリンタだけでなくスクリーンや補助出力装置へも出力可能です．最初に PRINT コマンドを起動すると，出力装置名を聞いてきますから，CON(スクリーン)，AUX(補助出力装置)または PRN(プリンタ)のどれかを指定します．以降 〝PRINT．COM〟 はメモリ中

に常駐し，PRINT コマンドを起動するたびに指定した出力装置へ出力されるようになります．

PRINT コマンドを使用するときには，次のようなことに注意しなければなりません．

(1)プリント中のファイルを含むディスケットは，プリントが終了するまでドライブから抜いてはいけません

(2)待ち行列に登録されているファイルは，変更／削除してはいけません．

(3) PRINT コマンド実行中に他のコマンドで PRINT コマンドの出力装置と同じ装置にデータを出力することはできません

サンプル・オペレーション

ディレクトリ FUNC にある〝．ASM〟の拡張子の付いたファイルをすべてプリント待ち行列に登録した後に，PRINT コマンドの各種スイッチの使用例を示します．

```
A>chdir func↵

A>dir *.asm↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

HEXOUT   ASM       529  84-02-06  16:36
FUNC-45  ASM       896  84-02-16  18:31
FUNC-25  ASM       768  84-02-15  17:28
AUXIN    ASM       185  84-02-19  21:13
       4 個のファイルがあります
  768000 バイトが使用可能です

A>print *.asm↵……拡張子が〝．ASM〟のファイルをすべてプリントアウトする

PRINT  Version 2.0

出力装置名を入力してください [PRN]:↵……出力装置名を入力する．この場合プリンタなのでリターンキーのみでよい（ここでCON，
                                        AUXなどを指定するとそれぞれスクリーン，補助出力に出力される）
ＰＲＩＮＴ．ＣＯＭ が常駐しました

        A:HEXOUT  .ASM を現在印刷中です
        A:FUNC-45 .ASM がプリント待ち行列に入っています
        A:FUNC-25 .ASM がプリント待ち行列に入っています
        A:AUXIN   .ASM がプリント待ち行列に入っています
                                        } 現在プリントアウトしているファイル名と待ち行列にはいっているファイル名を表示する

A>print func-25.asm /c hexout.asm↵……〝FUNC-25．ASM〟と〝HEXOUT．ASM〟のプリントアウトをキャンセルする．
                                        ／Cスイッチは，そのPRINTコマンドが終わるか，／Pが現れるまで有効
PRINT  Version 2.0

        A:FUNC-45 .ASM を現在印刷中です
        A:AUXIN   .ASM がプリント待ち行列に入っています
                                        } 〝HEXOUT．ASM〟と〝FUNC-25．ASM〟のプリントアウトはキャンセルされている

A>print /t↵……現在出力中，および待ち行列にはいっているファイルのプリントアウトをすべてキャンセルする

PRINT  Version 2.0

プリント待ち行列は空です

A>chdir ¥cal↵

A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥cal
```

```
.            <DIR>      84-02-23   1:40
..           <DIR>      84-02-23   1:40
FACT     PAS      279   84-02-23   1:51
SORT     PAS     2134   82-12-31  16:17
PRIMES   PAS      690   82-12-31  16:17
        5 個のファイルがあります
   768000 バイトが使用可能です

A>print fact.pas……"FACT.PAS" をプリントする

PRINT  Version 2.0

       A:FACT    .PAS を現在印刷中です

A>■
```

PROMPT

内部2.0

機能　MS-DOS のプロンプトを変更します.

書式　PROMPT ［〈プロンプト・テキスト〉］

解　説

PROMPT コマンドは，MS-DOS のプロンプトを変更するためのコマンドです．通常プロンプトは，カレント・ドライブを表す〝A>〟ですが，これを他のメッセージや普段使い慣れたターミナルのプロンプト，あるいは日付，時刻などの特別な意味を持ったプロンプトに変更することができます．また，エスケープ・シーケンスを用いてプロンプトに色を付けることも可能です．

プロンプトの変更には以下に示す文字列により指定できますが，機種によってはサポートされていない文字列もあります．

Table 1.3　プロンプトの変更と指定文字列

$t	現在の時刻
$d	現在の日付
$p	カレント・ドライブのカレント・ディレクトリ
$v	バージョン番号
$n	カレント・ドライブ
$g	文字〝>〟
$l	文字〝<〟
$b	文字〝|〟
$$	文字〝$〟
$_	改行(注：_はアンダースコア)
$s	空文字
$h	バックスペース
$e	ASCIIコードの1BH(エスケープコード)

なお，プロンプトをもとに戻すには次のように入力します．

PROMPT ↵　　あるいは　　PROMPT ng ↵

PROMPT コマンドによってプロンプトを設定すると，〝PROMPT＝〈プロンプト・テキスト〉〟の形でMS-DOSの環境に登録され，MS-DOSは環境の中からPROMPTを見つけて，それをプロンプトとして表示します．すなわち〝SET PROMPT＝［〈プロンプト・テキスト〉］〟としても同様な結果が得られます．また，バッチファイル中で〝%PROMPT%〟とすると設定されているプロンプト・テキストを参照することもできます．
(参考：エスケープ・シーケンス，SET，PATH)

サンプル・オペレーション

いくつかのプロンプトの設定例を示します．

```
A>prompt [MS-DOS] %↵……プロンプトを〝[MS-DOS]%〟に設定する
 └デフォルトのプロンプト
[MS-DOS] %prompt $n:↵……プロンプトを〝〈カレント・ドライブ〉:〟に設定する
 └プロンプト
A:prompt $d $t $$↵……プロンプトを〝〈日付〉〈時間〉$〟に設定する
 └プロンプト
1984-02-24 (金) 21:29:00.00 $prompt $v$_$p$_$n$g↵……プロンプトを〝〈バージョン番号〉
 └プロンプト                                          〈カレント・ディレクトリ〉
マイクロソフト MS-DOS バージョン  2.11                  〈カレント・ドライブ〉>〟に設定する
A:¥     └プロンプト
A>chdir subdir1↵……サブ・ディレクトリ〝SUBDIR1〟に移る

マイクロソフト MS-DOS バージョン  2.11
A:¥subdir1……〝¥SUBDIR1¥SUBDIR2〟になっている
A>chdir subdir2↵……サブ・ディレクトリ〝SUBDIR2〟に移る

マイクロソフト MS-DOS バージョン  2.11
A:¥subdir1¥subdir2……〝¥SUBDIR2〟になっている
A>prompt $l$p$b$n$g↵……プロンプトを　〝<〈カレント・ディレクトリ〉|〈カレント・ドライブ〉>〟に設定する

<A:¥subdir1¥subdir2|A>chdir ..↵……親ディレクトリに移る

<A:¥subdir1|A>chdir ..↵

<A:¥|A>prompt $e[7m$n$g$e[m↵……エスケープ・シーケンスを用いて〝A>〟を反転して表示する

A>prompt $n$g↵……プロンプトをもとに戻す

A>prompt $s↵……プロンプトを空文字にする

prompt↵……プロンプトをデフォルトに戻す

A>■
```

RECOVER

外部2.0

機能　不良セクタを含むファイルまたはディスクを修復します.
書式　RECOVER ［〈d：〉|〈ファイル名〉］

解　説

RECOVERコマンドは，ディスクに物理的な不良セクタがあった場合に，その不良部分を切り捨てることによってディスクを使用可能な状態に修復します．ディスクに傷が付いていたり，磁気的に破壊されていたりすると，そのディスクの不良部分の読み書きができなくなり，そこを読もうとするたびにエラーが出力されます．

このような場合，ディスクの一部が壊れている場合が多いため，その不良部分をFATから切り捨てることによってディスクをもとのように使えるようにします．

ディスクを修復するコマンドとしては，ほかにCHKDSKコマンドの／Fスイッチを付けたものがありますが，CHKDSK／Fが主にFATの整合性をチェックしてFATがソフトウェア的に壊されている場合にそれに応じた修復を行うのに対し，RECOVERコマンドはディスクの一部がハードウェア的に破壊された場合にその不良部分を切り捨てることによってディスクの修復をするということですから，この両者の修復の意味は本質的に異なっています．

RECOVERコマンドには大きく分けて2つの指定方法があります．

1つは，すべてのパラメータを省略するかドライブ名のみを指定する場合で，ルート・ディレクトリの不良セクタを修復するためのものです．これを実行するとディスク中のすべてのファイル名が〝FILE0001．REC〟，〝FILE0002．REC〟……という名前で，ルート・ディレクトリに集められます．

ルート・ディレクトリに十分な領域がないときには，メッセージを表示してFATにその情報を書き込みますから，COPYコマンドによって修復されたファイルを他のディスクへコピーした後に，DELコマンドによってそれらのファイルをディレクトリから削除するなどして，ルート・ディレクトリに十分な領域を確保します．その上で再度RECOVERコマンドを実行すれば，残りの分を修復できます．なお，〝RECOVER〈d：〉〟を，不良セクタを含んでいないディスクに対して実行しても同様にファイル名が変更され，ルート・ディレクトリに集められてしまうので，このコマンドを実行する場合には十分注意が必要です．

もう1つは，ファイル名を指定するもので，この指定を行うとそのファイル本体の不良セクタを修復します．この場合ファイル名は変更されませんが，その不良の程度によってはファイルの一部が切り捨てられることになります．ファイル名の指定にはワイルドカード(＊，？)も使用できますが，実際に修復されるのは最初に一致したファイルだけです．

RECOVERコマンドは，壊れたディスクを修復して使えるようにしますが，壊れたディスクが

完全にもとに戻るわけではなく，ただ単に不良セクタを切り捨てるだけなので，必要なファイルの一部あるいは全部が削られてしまうことがあります．ですから，RECOVERコマンドは壊れたディスクの中から被害をできるだけ最少限に食いとめるための最後の手段であると考えてください．日頃から大事なディスクはかならずバックアップをとるようにすべきです．

サンプル・オペレーション

実際に壊れてしまったディスクを，RECOVERコマンドを使って修復した例を示します．(参考：CHKDSK／F)

```
A>dir b:↵……壊れたディスクのルート・ディレクトリの内容を表示する(ドライブBにはいっている)

データエラーです <ドライブ B:> <読取り中>
中止<A>, もう一度<R>, 無視<I>? i
 ドライブ B: のボリュームラベルは MS-DOS HB
 ディレクトリは B:¥

COMMAND  COM    17190  83-09-19  16:19
KNJDIC   SYS    56320  83-09-20  14:22
BIN          <DIR>      84-02-22  20:32
COM          <DIR>      84-02-22  20:21
AUTOEXEC BAT       61   84-02-25  17:22
SORT     EXE     1360   83-09-04  23:54
OPR               89   84-02-23  17:52
OPR      TXT     2218   84-02-23  17:52
SUBDIR1      <DIR>      84-02-23   1:19
CAL          <DIR>      88-06-19   1:34
-Fo;%r ~E344635647  56-00-01   3:03
  @@@@@@ @@@328284224  12-02-00   8:02
  @@@@@@ @@@328284224  12-02-00   8:02
  @@@@@@ @@@328284224  12-02-00   8:02
  @@@@@@ @@@328284224  12-02-00   8:02
  @@@@@@ @@@328284224  12-02-00   8:02
  @@@@@@ @@@294729792  12-02-00   8:02

データエラーです <ドライブ B:> <読取り中>
中止<A>, もう一度<R>, 無視<I>? i
      17 個のファイルがあります
   860160 バイトが使用可能です

A>recover b:↵……ディレクトリを修復する

どれかキーを押してください．
ドライブ B: のファイルの修復を始めます↵……ここで何かキーを押すと修復がはじまる．まちがって壊れていないディスクを
                                              修復しないように入力待ちになり，メッセージが表示される

41 個のファイルを修復しました

A>dir b:↵……ルート・ディレクトリの内容を表示する

 ドライブ B: のディスクにはボリュームラベルがありません
 ディレクトリは B:¥

FILE0001 REC    40960  84-03-08  23:02
FILE0002 REC    18432  84-03-08  23:02
FILE0003 REC    17408  84-03-08  23:02
FILE0004 REC     1024  84-03-08  23:02
FILE0035 REC          84-03-08  23:02
FILE0036 REC     1024
FILE0037 REC    14336  84-03-08
FILE0038 REC     1024  84-03-08  23:02
FILE0039 REC     1024  84-03-08  23:02
FILE0040 REC     1024  84-03-08  23:02
FILE0041 REC     1024  84-03-08  23:02
      41 個のファイルがあります
   860160 バイトが使用可能です
```

```
A>for %c in (b:*.*) do recover %c ↵
A>recover B:FILE0001.REC

どれかキーを押してください.
ドライブ B: のファイルの修復を始めます

31744 バイト中 40960 バイトが修復されました

A>recover B:FILE0002.REC

どれかキーを押してください.
ドライブ B: のファイルの修復を始めます

15360 バイト中 18432 バイトが修復されました

A>recover B:FILE0041.REC

どれかキーを押してください.
ドライブ B: のファイルの修復を始めます

1024 バイト中 1024 バイトが修復されました

A>
A>chkdsk b:/v ↵
ディレクトリ B:
      B:¥FILE0001.REC
      B:¥FILE0002.REC
      B:¥FILE0003.REC
      B:¥FILE0040.REC
      B:¥FILE0041.REC

  1250304 バイト : 全ディスク容量
   342016 バイト : 41 個のユーザーファイル
    48128 バイト : 不良セクタ
   860160 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   187216 バイト : 使用可能メモリ

A>■
```

FORコマンドを使って，ドライブBのディスクのすべてのファイルを1つずつ修復する．"RECOVER B: *.*" としてもはじめのファイル（FILE 0001. REC）のみしか修復されない

不良セクタの分だけファイルが縮められている．明らかに前後の2つの数字はその順序が逆で，おそらくはメッセージのバグ．40960バイト中，31744バイトが修復されたと見るのが普通

不良セクタがなかったファイル

……すべてのファイルが修復された

……CHKDSK コマンドでディスクの状態を表示する

……RECOVERコマンドで修復したときに切り捨てられた不良セクタ

REN(RENAME)

内部1.25

機能　ファイル名を変更します．

書式　REN 〈ファイル名1〉〈ファイル名2〉

別表記　RENAME 〈ファイル名1〉〈ファイル名2〉

解　説

REN コマンドは，〈ファイル名1〉で指定されたファイル名を〈ファイル名2〉に変更します．

ファイル名の指定にはワイルドカード（*，?）を使うことができますが，〈ファイル名1〉でワイルドカードを用いると，それに適合するすべてのファイルが REN コマンドの対象となります．

また〈ファイル名2〉でワイルドカードを用いると，それに対応する位置の文字は変更されません．

〈ファイル名2〉がすでにそのディレクトリ内にある場合，すなわちすでに存在している名前に変更しようとすると，エラーメッセージが出力されファイル名は変更されません．

なお，RENコマンドを用いてディレクトリ名を変更することはできません(MS-DOSに，ディレクトリ名を変更するコマンドは用意されていません)．

サンプル・オペレーション

RENコマンドを用いて，実際にファイル名を変更します．

```
A>dir /w

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

.               ..              DUMP      ASM     DUMP      OBJ     DUMP      LST
FUNC-45  ASM    FUNC-45  COM    FUNC-45   LST     FUNC-25   ASM     FUNC-25   COM
FUNC-25  LST    AUXIN    ASM
       12 個のファイルがあります
   374784 バイトが使用可能です

A>ren func-45.asm dupltfh.asm ……ファイル名 "FUNC-45．ASM" を "DUPLTFH．ASM" に変更する

A>ren auxin.asm con??.?bc ……ファイル名 "AUXIN．ASM" を "CONIN．ABC" に変更する．新ファイル名(第2パラメータ)にワイルドカード("＊"または"？")を用いるとその部分は変更されない

A>dir /w

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func

.               ..              DUMP      ASM     DUMP      OBJ     DUMP      LST
DUPLTFH  ASM    FUNC-45  COM    FUNC-45   LST     FUNC-25   ASM     FUNC-25   COM
FUNC-25  LST    CONIN    ABC
       12 個のファイルがあります
   374784 バイトが使用可能です

A>dir ¥cal¥fact.pas /w

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥cal

FACT     PAS
       1 個のファイルがあります
   374784 バイトが使用可能です

A>rename ¥cal¥fact.pas factorial.pas ……サブ・ディレクトリ "CAL" にある "FACT．PAS" を "FACTORIAL．PAS" にファイル名を変更する．RENAMEはRENの別表記

A>dir ¥cal¥fact*.pas /w

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥cal

FACTORIA PAS ……ファイル名が変更された．ただし "FACTORIAL" は9文字で1文字はみ出してしまっているのでファイル名は "FACTORIA．PAS" となっている
       1 個のファイルがあります
   374784 バイトが使用可能です

A>ren *.lst *.prn ……拡張子が "．LST" のファイル名を拡張子だけ "．PRN" に変更する

A>dir /w

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥func
```

```
.           ..                    DUMP      ASM     DUMP      OBJ     DUMP      PRN
DUPLTFH  ASM    FUNC-45  COM      FUNC-45   PRN     FUNC-25   ASM      FUNC-25   COM
FUNC-25  PRN    CONIN    ABC
        12 個のファイルがあります
    374784 バイトが使用可能です

A>■
```

RMDIR(RD)

内部２．０

機能　ディスクから指定されたサブ・ディレクトリを削除します．

書式　RMDIR〈パス名〉

別表記　RD〈パス名〉

解　説

RMDIR コマンドは，MKDIR，CHDIR などとともに MS-DOS の階層的ディレクトリ構造をサポートするためのコマンドで，不要になったサブ・ディレクトリを階層的ディレクトリ構造から削除します．〝.〟ディレクトリと〝..〟ディレクトリ以外のファイルやディレクトリが存在すると，そのディレクトリを削除することはできません．これは誤まってそのディレクトリの下にある必要なファイルやディレクトリを削除してしまうことを防ぐためです．

〈パス名〉によってディレクトリを指定するには，ルート・ディレクトリからの絶対指定，カレント・ディレクトリからの相対指定が利用できます．ただし，カレント・ディレクトリは削除できませんから，それより上の階層のディレクトリから RMDIR コマンドを実行しなければなりません．

(参考：CHDIR，MKDIR)

サンプル・オペレーション

RMDIR コマンドを使って，ディレクトリの削除を行います．最初のディレクトリ構造は Fig. 1. 7 のようになっているものとします．

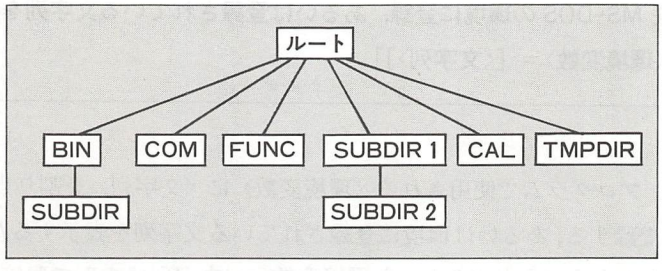


Fig. 1．7　ディレクトリの状態

```
A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
KNJDIC   SYS     56320  83-09-20  14:22
         <DIR>           84-02-22  20:32
                         84-02-23  17:52
OPR      TXT
SUBDIR1      <DIR>       84-02-
CAL          <DIR>       84-02-23
TMPDIR       <DIR>       84-02-23   4:36 ……削除するディレクトリ
       12 個のファイルがあります
   796672 バイトが使用可能です

A>rmdir tmpdir↵……サブ・ディレクトリ "TMPDIR" を削除する

A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
KNJDIC   SYS     56320  83-09-20  14:22
         <DIR>           84-02-22  20:32
COM
OPR      TXT      2218
SUBDIR1      <DIR>       84-02-
CAL          <DIR>       84-02-23   1:40
       11 個のファイルがあります
   797696 バイトが使用可能です

A>rd subdir1¥subdir2↵……サブ・ディレクトリ "SUBDIR1" にあるサブ・ディレクトリ "SUBDIR2" を削除する.
                        RDはRMDIRの別表記
A>dir subdir1↵……サブ・ディレクトリ "SUBDIR1" の内容を表示する

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥subdir1

.            <DIR>       84-02-23   1:19
..           <DIR>       84-02-23   1:19   サブ・ディレクトリ "SUBDIR2" は削除されている
       2 個のファイルがあります
   798720 バイトが使用可能です

A>rmdir cal↵……サブ・ディレクトリ "CAL" を削除する
パスの指定が違うか，ディレクトリでないか，   "CAL"の中にサブ・ディレクトリまたはファイルがあるので
またはディレクトリが空ではありません         "CAL" は削除できない

A>■
```

SET

内部2.0

機能　文字列をMS-DOSの環境に登録，あるいは登録されている文字列を表示します．

書式　SET　[〈環境変数〉＝［〈文字列〉］]

解　説

SETコマンドは，プログラムで使用される〈環境変数〉に〈文字列〉を割り当て，それをMS-DOSの環境領域に登録する，あるいは環境に登録されている文字列を表示するためのものです．

SETコマンドで次のように入力すると，"〈環境変数〉＝〈文字列〉"の形で環境に登録されます．

A > SET 〈環境変数〉=〈文字列〉 ↵

これ以降，アプリケーション・プログラムやコマンドは〈文字列〉を〈環境変数〉を通して参照できるようになりますから，プログラム自体を直接変更することなしにSETコマンドで〈環境変数〉に割り当てる〈文字列〉を変更するだけで，その中で使う文字列を容易に変更することができます．

バッチファイル中でも環境の文字列を参照することができますが，バッチファイルの中で〝%〈環境変数〉%〟とすると，その部分が実行時にはそれに割り当てられた〈文字列〉に置き換えられます．

〈文字列〉を省略して次のように入力した場合，〈環境変数〉に割り当てられている〈文字列〉が環境中から削除されます．

SET 〈環境変数〉=↵

パラメータを省略して単にSETのみを入力すると，現在環境に登録されている〈環境変数〉とそれに割り当てられた〈文字列〉を表示します．

なお，PATHコマンドやPROMPTコマンドで，パスあるいはプロンプトを設定すると，それらは環境領域の中に登録されます．つまり，PATHコマンド，PROMPTコマンドは〝COMMAND．COM〟が使用する環境を設定するために設けられたSETコマンドの特別なものということができ，これらのコマンドによって設定された環境は，SETコマンドによるものとまったく同じように扱うことができます．

たとえば，バッチファイル中で〝%PATH%〟とすると，現在設定されているパスを参照することができます．

MS-DOSのブート時には，コマンド・プロセッサ(通常は〝COMMAND．COM〟)を再ロードするためのパスおよびコマンド・プロセッサ名を表す〝COMSPEC=〈パラメータ〉〟が，自動的に環境に登録されます．

(参考：PATH，PROMPT，〝CONFIG．SYS〟)

サンプル・オペレーション

バッチファイルを用いたLOGINコマンド(ハードディスクのシステムなどで，複数のユーザーがシステムを共有している際に用いられるユーザーの識別コマンド)およびHOMEコマンド(各ユーザーごとに設定した起点となるディレクトリに戻るコマンド)の作成を示します．このLOGINコマンドは環境変数を設定し，ユーザーのディレクトリに移るというものですが，LOGINコマンドの最後に／Dスイッチを付けるとSETコマンドでLOGINDIRに設定しておいたユーザーのサブ・ディレクトリに，それを省略するとユーザーのホーム・ディレクトリに移るというものです．SETコマンドでLOGINDIRを他のサブ・ディレクトリに設定すれば，簡単

に他のサブ・ディレクトリにもLOGINできるようになります.

また，HOMEコマンドはユーザーのサブ・ディレクトリからホーム・ディレクトリへ戻るというものです.

なお，ディレクトリ構造はFig. 1.8のようになっています.

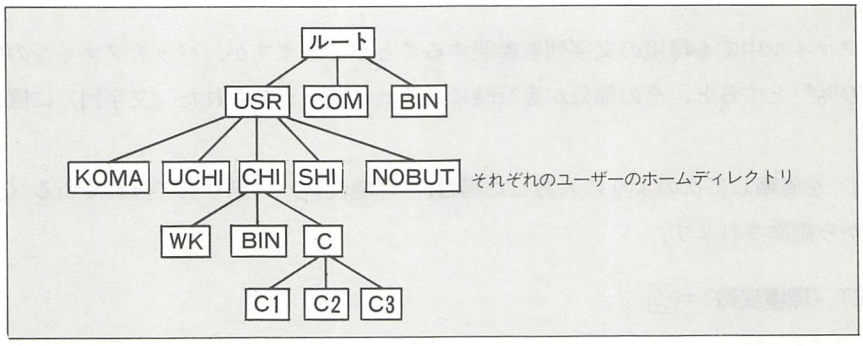


Fig. 1.8　ディレクトリの状態

```
A>set↵……環境に登録されている文字列を表示する
COMSPEC=¥COMMAND.COM ……MS-DOSブート時に自動的に登録される
PATH=A:¥;A:¥COM;A:¥BIN ……"AUTOEXEC.BAT" によって自動設定している

A>type login.bat↵……LOGINコマンドをタイプアウト
:login
  echo off
  if '%1'=='' goto error ……ユーザー名が入力されなかったらエラー

  cls
  echo ***  Welcome to PC-9801 MS-DOS, Mr.%1  ***

  set user=%1 ……USERにユーザー名をセットする
  set home=¥usr¥%user% ……HOMEにユーザーのホーム・ディレクトリのパスを設定する
  path a:¥;a:¥com;a:¥bin ……パスを設定する
  set prompt=$n$g ……プロンプトを設定する. "PROMPT $n$g"でも同じ

  chdir %home%
  if '%2'=='/d' chdir %logindir% ……/Dスイッチが指定されていたらLOGINDIRに設定されたディレクトリに移る
                                   たとえば "SET LOGINDIR=BIN" と設定しておくと実行時 "%LOGINDIR%" はBINに置き換えられる
  goto end
:error
  echo ***  Usage : login 'user name'  ***
:end

A>type home.bat↵……HOMEコマンドをタイプアウト
chdir %home% ……ユーザーがCHDIRコマンドでどのディレクトリへ移っていてもHOMEを入力するとホーム・ディレクトリに戻れる

A>set logindir=c¥c1↵……LOGINDIRを "C¥C1" に設定する

A>set↵
COMSPEC=¥COMMAND.COM
PATH=A:¥;A:¥COM;A:¥BIN
LOGINDIR=c¥c1 ――――― 今SETコマンドで登録したLOGINDIR

A>login chi /d↵……/Dスイッチを付けてユーザーディレクトリCHIにログインする. この場合¥USR¥CHI¥C¥C1にカレント
                  ディレクトリが変更される
A>  echo off
***  Welcome to PC-9801 MS-DOS, Mr.chi  ***

A>chdir↵
A:¥usr¥chi¥c¥c1 ……¥USR¥CHI¥C¥C1に移っている
```

```
A>home↵ ……ホーム・ディレクトリ(¥USR¥CHI)に移る

A>chdir ¥usr¥chi ……HOMEコマンド(バッチファイル)が自動的に出力する

A>
A>dir↵

 ドライブ A: のボリュームラベルは PC-9801
 ディレクトリは A:¥usr¥chi

.            <DIR>      84-02-25  14:58
..           <DIR>      84-02-25  14:58
C            <DIR>      84-03-19   7:42
BIN          <DIR>      84-03-19   8:03
WK           <DIR>      84-03-19   8:10
       5 個のファイルがあります
   792576 バイトが使用可能です

A>set↵
COMSPEC=¥COMMAND.COM
LOGINDIR=c¥c1
USER=chi ……ユーザー名
HOME=¥usr¥chi ……ユーザーのホーム・ディレクトリ  } LOGINコマンドによって設定された
PATH=A:¥;A:¥COM;A:¥BIN
PROMPT=$n$g

A>■
```

SORT

外部2.0

機能　標準入力から読み込んだデータをソートして標準出力に出力します.

書式　SORT　[／R] [／＋n]

解　説

SORT コマンドはフィルタの一種で，標準入力から読み込んだデータ(一般にはファイル)の各行を，指定した桁についてキャラクタコード順に並べ換えるコマンドです．一般に，ファイルの内容や DIR コマンドの出力などは，その並びに規則性がないために，これらの情報は人間にとってはたいへん理解しにくいものです．SORT コマンドでこれらの情報をある規則(キャラクタコード順)に沿って並べ換えることによって，より理解しやすいものにすることができます．

SORT コマンドには次の 2 つのスイッチがあります．

／R　　逆順ソートを指定します．アルファベットならばZからAの順で並換えを行います．

／＋ n　ソートの対象とする文字(キー)を n 桁目以降に指定します．このスイッチを省略すると，1桁目すなわち行のはじめからの文字で並換えを行います．

SORT コマンドは，MORE コマンド同様，その入力は標準入力(キーボード)，出力には標準出力(スクリーン)しか許されません．もしファイルの内容や DIR コマンドの出力を SORT コマンドの入力にする場合には I／O のリダイレクトあるいはパイプ処理を用いることになります．

MS-DOS における漢字処理は，すべてシフト JIS コード(マイクロソフト漢字コード)によって処理されているために，SORT コマンドの入力に漢字が含まれている場合には，かならずしも

漢字の読みの順には並べ換えられません.

(参考： MORE, FIND)

サンプル・オペレーション

SORT コマンドを利用して,EDLIN で作ったファイルや,DIR コマンドの出力を並べ換えた例を示します.

```
A>edlin language.txt↵……EDLINで "LANGUAGE . TXT" を作る
新しいファイルです
*i↵……インサート・コマンド
     1:*MS-BASIC↵
     2:*MS-BASCOM↵
     3:*GW BASIC↵
     4:*GW BASCOM↵
     5:*MS-FORTRAN↵
     6:*MS-Pascal↵
     7:*MS-COBOL↵
     8:*JANUS/ADA↵
     9:*Lattice C compiler with C-FOOD↵
    10:*mbp COBOL↵
    11:*muLISP/muSTAR↵
    12:*Prolog-J86↵
    13:*APL/8086↵
    14:*^Z↵…… CTRL - Z ↵を入力してインサートモードから抜ける
*e↵……EDLINを終了しDOSのコマンドレベルへ戻る

A>sort < language.txt↵……今作った "LANGUAGE . TXT" をソートする （ファイルをSORTの標準入力にする)MS-DOSのSORTコマンドはSORT〈ファイル名〉の型では使えない. ファイルの内容をソートしたいときは"SORT < 〈ファイル名〉"もしくは"TYPE〈ファイル名〉 | SORT"とする
APL/8086
GW BASCOM
GW BASIC
JANUS/ADA
Lattice C compiler with C-FOOD
MS-BASCOM
MS-BASIC
MS-COBOL
MS-FORTRAN
MS-Pascal
Prolog-J86
mbp COBOL
muLISP/muSTAR

A>chdir com↵

A>dir | sort↵……ディレクトリの内容をソートして出力する(DIRコマンドの出力をSORTに送る)

       24 個のファイルがあります
   793600 バイトが使用可能です
 ディレクトリは A:¥com
 ドライブ A: のボリュームラベルは MS-DOS HB
.           <DIR>      84-02-22  20:21
..          <DIR>      84-02-22  20:21
ASM      BAT        66  84-02-21  15:28
ASSIGN   COM      1069  83-08-12  14:32
                    65  84-02-22   3:31
SORT     EXE          -09-04  23:54
SPEED    COM      120?          15:07
SWITCH   COM      2054  83-10-2?
SYS      COM       968  83-09-04  23:57
USKCGM   COM      4130  83-10-30  19:43
                                          } 先頭の文字から，キャラクタコード順にソートされる

A>dir *.com | sort /r↵……ディレクトリ中，拡張子が ". COM" のファイルを逆順ソートして出力
USKCGM   COM      4130  83-10-30  19:43
SYS      COM       968  83-09-04  23:57
                  2054  83-10-28  16:30
SPEED    COM          -08-10  15:07
RECOVER  COM      260?          23:51
ASSIGN   COM      1069  83-08-
 ドライブ A: のボリュームラベルは MS-
 ディレクトリは A:¥com
   794624 バイトが使用可能です
       15 個のファイルがあります
                                          } 先頭の文字から，キャラクタコードの逆順にソートされる
```

```
A>dir | sort /+24 | more.....ディレクトリの内容の左から24番目(年月日)以降をキーとしてソートし，さらにその結果をMOREに
                            送り１画面ずつ出力する

 ディレクトリは A:¥com
                                  ←これ以降でソート．ファイルの作成年月日順に並ぶ
SPEED    COM      1209  83-08-10  15:07
ASSIGN   COM      1069  83-08-12  14:32
                 ~1224  83-08-25  13:52
FORMAT   COM           ~10-29  15:46
USKCGM   COM      4130  ~      ~43
KEY      COM      4531  83-10-30
-- More -- .....何かキーを押すと次が表示される
ASM      BAT        66  84-02-21  15:28
CC       BAT        65  84-02-22   3:31
.            <DIR>      84-02-22  20:21
..           <DIR>      84-02-22  20:21
 ドライブ A: のボリュームラベルは MS-DOS HB
      24 個のファイルがあります
  793600 バイトが使用可能です

A>■
```

SYS

外部１.25

機能　カレント・ドライブの MS-DOS のシステムファイル(〝IO．SYS〞，〝MSDOS．SYS〞)を〈d：〉で指定したディスクへコピーします．

書式　SYS 〈d：〉

解　説

SYS コマンドは，隠された２つのシステムファイル(〝IO．SYS〞，〝MSDOS．SYS〞)を〈d：〉で指定されたドライブのディスクへコピーするもので，システムファイルを更新したり，フォーマット済みのディスクにシステムファイルをコピーするためのコマンドです．

〈d：〉で指定された受け側のディスクは，完全に空か，転送しようとしているシステムファイルとまったく同じ大きさの〝IO．SYS〞，〝MSDOS．SYS〞が存在していなければなりません．

なお，SYS コマンドでは〝COMMAND．COM〞をコピーしないので，これを転送するときには，COPY コマンドを用います．SYS コマンドを実行し，その後〝COMMAND．COM〞をコピーすれば，それ以降そのディスクは MS-DOS のブート用に使用できるようになります．ただし，フォーマットの済んでいない新しいディスクでシステムディスクを作る場合には，FORMAT／S コマンドを利用すれば，フォーマット終了後にシステム(〝IO．SYS〞，〝MSDOS．SYS〞)および〝COMMAND．COM〞を自動的に転送するので最も簡単にシステムディスクを作ることができます．

(参考：FORMAT)

サンプル・オペレーション

フォーマット済みのディスクを，MS-DOSがブートできるシステムディスクにします．まず，SYSコマンドを用いて"IO．SYS"と"MSDOS．SYS"をコピーし，次にCOPYコマンドで"COMMAND．COM"をコピーしています．

```
A>chkdsk b: /v↵……ドライブBのディスクの状態を調べる
ディレクトリ B:  }まったく何のファイルもはいっていない

  1250304 バイト : 全ディスク容量
  1250304 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   198672 バイト : 使用可能メモリ


A>sys b:↵……システム"IO．SYS"と"MSDOS．SYS"をドライブBのディスクに転送する
システムが転送されました

A>chkdsk b: /v↵
ディレクトリ B:
     B:¥IO.SYS      }システムが転送されている
     B:¥MSDOS.SYS

  1250304 バイト : 全ディスク容量
    59392 バイト : 2 個のシステムファイル
  1190912 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   198672 バイト : 使用可能メモリ


A>copy command.com b:↵……"COMMAND．COM"をドライブBにコピーする
        1 個のファイルをコピーしました

A>dir b:↵

 ドライブ B: のディスクにはボリュームラベルがありません
 ディレクトリは B:¥

COMMAND  COM    17190  83-09-19  16:19  }"COMMAND．COM"がコピーされた
      1 個のファイルがあります            }以上でMS-DOSのシステムディスクができた
  1173504 バイトが使用可能です

A>■
```

TIME　　内部1．25

機能　システムが管理する時刻の表示／設定を行います．

書式　TIME　[〈hh〉[：〈mm〉[：〈ss〉]]]

解　説

MS-DOSは，ファイルを作成または更新した際にその日付と時刻をディレクトリに記録しますが，このコマンドによってその時刻を管理しています．MS-DOSがブートされた際に"AUTOEXEC．BAT"が存在しなければ，DATEコマンドに続いて起動されます．

各数字の範囲は次のとおりです．

〈hh〉　：00～23（時）
〈mm〉　：00～59（分）
〈ss〉　：00～59（秒）

〝AUTOEXE．BAT〟が存在している場合は，TIMEコマンドおよびDATEコマンドは実行されませんから，クロック・デバイスを持たない機種で〝AUTOEXEC．BAT〟を使用する場合には，これらのコマンドをあらかじめ〝AUTOEXEC．BAT〟ファイルの中に入れておくとよいでしょう．
（参考：DATE）

サンプル・オペレーション

TIMEコマンドを用いて時刻を設定した例を示します．

```
A>time 13:00↵……時間を13：00に設定する．この場合13あるいは13：00：00でもよい

A>time↵……TIMEのみを入力すると現在の時刻を表示し，時刻の入力待ちになる
現在の時刻は 13:00:03.00 です
時刻を入力してください：↵……この場合は時刻を変更しないので↵のみを入力する

A>■
```

TYPE　　内部1．25

機能　ファイルの内容をスクリーンに表示します．
書式　TYPE　[〈d：〉][〈パス名〉]〈ファイル名〉

解　説

TYPEコマンドは〈ファイル名〉で指定したテキストファイルの内容をスクリーンに表示するためのコマンドです．

TYPEコマンドで，アスキーファイル以外のバイナリファイルをスクリーンに表示するとコントロールコードをスクリーンに送ることになるので，正しい表示は行われません．

TYPEコマンドの本来の機能は単純なものですが，これをリダイレクトやデバイスファイルとともに用いると，いろいろと応用することができます．

サンプル・オペレーション

最初の例では〝AUTOEXEC．BAT〟ファイルを作り，そのファイルをタイプアウトしています．

次の例では，既存のファイルにテキストを追加(アペンド)しています．EDLINなどのエディタ

を使ってテキストを作った場合，通常(EDLINの場合はかならず)ファイルの最後に自動的にEOF(CTRL-Z：1AH)が付けられます．したがってその後ろにTYPEコマンドを使ってテキストをアペンドしても，EOFが途中にはさまれてしまい，それ以降の文字列，すなわちアペンドしたテキストが無効となります．つまりそのファイルをTYPEコマンドでスクリーンに表示しようとしても，追加部分は無視されて出力されません．

ところが，次のような形式でファイルを作成した場合には，作成したファイルの後ろにはEOFが付かないので，TYPEコマンドを使ってテキストを正しくアペンドすることができます．

TYPE　CON > 〈ファイル名〉 ↵　あるいは　COPY　CON 〈ファイル名〉 ↵

```
A>type con > autoexec.bat ……キーボードからの入力をファイル "AUTOEXEC.BAT" におとす．小さなファ
echo off                    イルならばこの方法で簡単に作成できる．"COPY CON 〈ファイル名〉"とほぼ同じ
date
time
path a:¥;a:¥com;a:¥bin
^Z …… CTRL - Z を入力してTYPEを終わる(CTRL-Zはファイルの終わり(EOF)の意味)

A>type autoexec.bat ……今作った "AUTOEXEC.BAT" を表示する
echo off
date
time
path a:¥;a:¥com;a:¥bin

A>edlin asm.bat ……EDLINで "ASM.BAT" を作る
新しいファイルです
*i ……インサートモードにはいる
     1:*masm %1,,%1;
     2:*link %1;
     3:*exe2bin %1 %1.com
     4:*^Z
*e ……EDLINを終了する

A>type asm.bat > prn ……今作ったファイル "ASM.BAT" をプリンタへ出力する

A>type con >> autoexec.bat ……前に作った "AUTOEXEC.BAT" にキーボードからテキストを
speed r0 9600 b8 pn s1 xon    アペンドする ">>" はアペンドの意味
^Z

A>type autoexec.bat
echo off
date
time                                  } アペンドされた
path a:¥;a:¥com;a:¥bin
speed r0 9600 b8 pn s1 xon
          └アペンドされたテキスト
A>dir a*

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

AUTOEXEC BAT       74  84-02-25   1:41
ASM      BAT       44  84-02-25   1:38
       2 個のファイルがあります
   824320 バイトが使用可能です

A>type con >> asm.bat ……今度はEDLINで作ったファイル "ASM.BAT" にキーボードからテキストをアペンドする
%1
^Z

A>type asm.bat ……"ASM.BAT" をタイプアウトする
masm %1,,%1;
link %1;                  } アペンドされていない(本来ならば最終行に "%1" がアペンドされているはず)
exe2bin %1 %1.com
```

```
A>dir asm.*

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

ASM      BAT        48  84-02-25   1:43      ……確かにファイルサイズは増えているようである
       1 個のファイルがあります
  824320 バイトが使用可能です

A>dump asm.bat ……"ASM.BAT" をダンプしてみる(DUMPコマンドはPC-9800シリーズ専用のコマンド)

Dump Version 2.0


00000000  6D 61 73 6D 20 25 31 2C-2C 25 31 3B 0D 0A 6C 69   masm %1,,%1;..li
00000010  6E 6B 20 25 31 3B 0D 0A-65 78 65 32 62 69 6E 20   nk %1;..exe2bin
00000020  25 31 20 25 31 2E 63 6F-6D 0D 0A 1A 25 31 0D 0A   %1 %1.com...%1..
                                            ↑アペンドする前に1AH(EOF)がはいっていた. 1AHはMS-DOSではアスキーファイルの終わりを意味するのでそれ以降のデータ(アペンドしたテキスト)は無効となっている

A>edlin asm.bat /b ……EDLINを／Bスイッチを付けて起動する途中の1AH(EOF)をテキストの内容としてエディットできるようになる
ファイルを終りまで読込みました
*1,#1
     1:*masm %1,,%1;
     2: link %1;
     3: exe2bin %1 %1.com
     4: ^Z%1
*4 ……第4行をエディットする
     4:*^Z%1 ……このCTRL-Zを取り除く
     4:*%1
*e ……EDLINを終了する

A>type asm.bat
masm %1,,%1;
link %1;
exe2bin %1 %1.com
%1

A>■
```

今度はきちんと最終行まで表示された

VER

内部2.0

機能　現在使用中の MS-DOS のバージョン番号を表示します.

書式　VER

解　説

現在使用している MS-DOS のバージョン番号を表示します. VER コマンドは，ファンクション 30H のシステムコールによって，MS-DOS のバージョン番号を表示しているだけです.

コマンドやアプリケーション・プログラムでは，DOS のバージョンチェックを行うことによって，そのプログラムが実行できるかどうかの確認に利用しますが，DOS のコマンドレベルではこの VER コマンドはあまり意味を持ちません.

サンプル・オペレーション

VER コマンドを使って MS-DOS のバージョンを調べています. この場合 2.11 と表示されます.

```
A>ver↵……MS-DOSのバージョンを調べる
ﾏｲｸﾛｿﾌﾄ MS-DOS ﾊﾞｰｼﾞｮﾝ  2.11
A>■
```

VERIFY

内部2.0

機能　ディスクへ書込みを行う際，ベリファイ(検査)を行うかどうかを設定／表示します．

書式　VERIFY　[ON | OFF]

解　説

VERIFY コマンドは，ディスクに書き込む際にベリファイ(ファイルが正しく書き込まれているかチェックする)をするか否かを設定／表示します．VERIFY を ON にすると，ディスクへ書き込むごとにベリファイを行いますが，VERIFY を OFF にするとベリファイは行いません．パラメータを省略して VERIFY のみを入力すると，現在の設定状況が表示されます．

VERIFY の ON／OFF の状態は，VERIFY コマンドを使って変更するか，プログラム中で(ファンクション 2EH のシステムコール SET VERIFY を使って)変更するまで変わりません．

MS-DOS ブート時には VERIFY OFF になっていますが，重要なデータをディスクに書き込む際には ON にした方がよいでしょう．ただし，ディスクエラーはまれですし，ベリファイを行った場合は書込みに時間がかかるので，通常は OFF のままでかまいません．

サンプル・オペレーション

VERIFY コマンドを使って VERIFY を設定してみます．

```
A>verify↵……ベリファイの設定状況を調べる．MS-DOS起動時はVERIFY OFF
VERIFY は <OFF> です

A>verify on↵……ベリファイをONにする．これ以降ディスクにデータを書き込むごとに，きちんとディスクに書き込まれたかどうかチェックするようになる
A>verify↵
VERIFY は <ON> です

A>verify off↵……ベリファイをOFFにする

A>verify↵
VERIFY は <OFF> です

A>■
```

VOL

内部2．0

機能　ディスクのボリュームラベルを表示します．

書式　VOL ［〈d：〉］

解　説

VOLコマンドは，ドライブ名〈d：〉で指定されたディスクのボリュームラベルを表示します．ドライブ名を指定しないと，カレント・ドライブのディスクのボリュームラベルを表示します．

ボリュームラベルの設定にはFORMAT／Vコマンドを使用します．なおボリュームラベルは，ディスクをフォーマットしない限りCOMMANDからは変更できません．

（参考：FORMAT）

サンプル・オペレーション

実際にVOLコマンドを用いてディスクのボリュームラベルを表示してみます．ドライブBのディスクにはボリュームラベルが設定してないので，その旨メッセージが出力されています．

```
A>vol↵……カレント・ドライブのボリュームラベルを表示する
 ドライブ A: のボリュームラベルは MS-DOS HB
A>vol b:↵……ドライブBのボリュームラベルを表示する
 ドライブ B: のディスクにはボリュームラベルがありません
A>■
```

バッチコマンド

ECHO

内部 2.0

機能　バッチファイル実行中にコマンドを表示するかどうかを設定／表示します．また ECHO コマンドに続くメッセージを標準出力へ出力します．

書式　ECHO ［ON | OFF | 〈メッセージ〉］

解　説

ECHO　ON に設定すると，バッチファイル実行中にそのバッチファイル中のコマンドを実行する際，そのコマンドをスクリーンに表示(エコーバック)させ，ECHO OFF に設定すると，バッチファイル実行中コマンドをスクリーンに表示しなくなります．

バッチファイル実行中，コマンドが表示されては見にくくなるような場合に ECHO OFF をバッチファイルの最初で実行しておけば，それ以降のコマンドは表示されなくなります．

ECHO コマンドに続けてメッセージを書くと，それを標準出力へ出力することができますが，これはバッチファイル実行中に何かメッセージを表示させたり，リダイレクトを利用するような場合に用います．また，バッチファイル中でこの ECHO 〈メッセージ〉 を使った場合は，メッセージ中のパラメータ(%0～%9)にはそれに対応する値が割り当てられます．ただし，この場合 ECHO ON のままにしておくと，ECHO 〈メッセージ〉 とその実行結果 〈メッセージ〉 が出力されますから，ECHO OFF にしておくとよいでしょう．

ECHO のみを入力すると，現在の ECHO の状態(ON または OFF)を表示します．

なお，MS-DOS ブート時には ECHO ON に設定されています．また，キーボードから ECHO OFF を入力すると，それ以降 ECHO ON に設定されるかあるいはバッチ処理が終わって自動的に ECHO ON に設定されるまでは，プロンプトが表示されなくなります．

サンプル・オペレーション

環境変数を用いてバッチファイルのメッセージを出力する例を示します．この例では，ECHO コマンドは主にメッセージの表示に使われています．

```
A>type tstecho.bat
echo on
rem  message
echo %0.bat is sample for echo command
echo %os% is running
```

```
echo off
rem  message
echo %0.bat is sample for echo command
echo %os% is running

A>set os=MS-DOS↵……環境変数"OS"に文字列"MS-DOS"を代入する

A>tstecho↵……バッチファイル "TSTECHO" を実行する

A>echo on

A>rem  message
          └──"%0" には実行中のバッチファイル名が割り当てられている
A>echo tstecho.bat is sample for echo command
tstecho.bat is sample for echo command

A>echo MS-DOS is running
MS-DOS is running
└環境変数"OS"にセットされている文字列，バッチファイル中で"%OS%"で参照できる
A>
A>echo off……ECHOをOFFに
tstecho.bat is sample for echo command
MS-DOS is running

A>echo↵……バッチ処理終了．現在のECHOの状態を表示する
ECHO は <ON> です
          └バッチ処理終了後は自動的にONになる
A>echo off↵……ECHOをOFFにする
echo↵……プロンプトが表示されなくなる．現在のECHOの状態を表示する
ECHO は <OFF> です
echo on↵

A>■
```

ECHO ONなので，実行するコマンドが表示されている．ECHO〈メッセージ〉はコマンドとその実行結果の〈メッセージ〉が二重に表示されて見苦しくなっている

ECHO OFFなので，コマンドは表示されないで〈メッセージ〉のみが表示されている

FOR

内部2．0

機能　〈集合〉中のそれぞれの要素に対して〈コマンド〉で指定された処理を実行します．

書式　FOR%%〈変数〉 IN(〈集合〉)DO 〈コマンド〉 …バッチ処理(バッチファイル中)

　　FOR%〈変数〉 IN(〈集合〉)DO 〈コマンド〉 …　逐次処理用(キーボードから入力)

解　説

FORコマンドは，いくつかのアセンブラ・ソースファイルをまとめてアセンブルしたり，不要なファイルをまとめて削除するといった複数のファイルに対して，ある特定のコマンドを実行する場合に用いられます．ファイル名にワイルドカードを使えるほどの共通性がない場合，あるいはTYPEコマンドのようにワイルドカードが使えないコマンドでは，FORコマンドによる一括処理が行えます．

FORコマンドは〈集合〉の要素を順々に%〈変数〉に代入し，〈コマンド〉を解釈して実行します．

ここで〈変数〉はバッチ処理におけるパラメータと区別するために0～9以外の任意の文字を用います．ただし，FORの次の〈変数〉とコマンド中の変数は対応していなければなりません．

〈集合〉の要素にはワイルドカードを指定することができますが，要素中に1つでもワイルドカ

ードを使って指定したものがあると，集合のいちばん最初の要素のみが有効となり，それ以外は無視されます．また，最初の要素がワイルドカードを使って指定された場合のみ，それに一致するすべてのファイル名がその対象となります．

FORコマンドは2通りの書式があり，バッチファイル中でFORコマンドを使うときのみ，バッチ処理用の書式を用いますが，それ以外，たとえばキーボードからFORコマンドを入力するときには下の逐次処理用の書式を用います．これは，バッチファイル中のパラメータ(%0～%9)と区別するためで，この書式を間違えるとFORコマンドは正しく実行されません．

サンプル・オペレーション

パラメータに指定された拡張子を持つファイルをすべて削除するバッチファイルを作成します．

```
A>copy con kill.bat
for %%f in (%1 %2 %3 %4 %5) do del *.%%f
^Z
        1 個のファイルをコピーしました

A>chdir asmwork

A>dir

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥asmwork

.            <DIR>      84-02-25   7:22
..           <DIR>      84-02-25   7:22
FUNC-00  ASM       768  84-02-20  21:54
FUNC-03  ASM       768  84-02-20  21:55
FUNC-06  ASM       512  84-02-20  21:56
FUNC-0A  ASM      1152  84-02-20  21:57
                        84-02-20  21:59
                        84-02-25   8:30
FUNC-06  COM
FUNC-0B  OBJ       226  84-02-
FUNC-0B  LST      3017  84-02-25   8:2
FUNC-0B  EXE       891  84-02-25   8:21
FUNC-0B  COM       123  84-02-25   8:21
      20 個のファイルがあります
  740352 バイトが使用可能です

A>kill obj exe

A>for %f in (obj exe    ) do del *.%f

A>del *.obj
A>del *.exe

A>
A>
A>dir

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥asmwork

.            <DIR>      84-02-25   7:22
..           <DIR>      84-02-25   7:22
FUNC-00  ASM       768  84-02-20  21:54
             768  84-02-20  21:55
                  84-02-20  21:56
ASM      HIS
FUNC-06  LST      1375  84-02-
FUNC-06  COM        30  84-02-25   8:2
FUNC-0B  LST      3017  84-02-25   8:21
```

……COPYコマンドによって指定された拡張子（最大5つまで）を持つすべてのファイルを削除するバッチファイル を作る

バッチファイル中でFOR文を書くときには，%を2つ重ねる

0～9以外の文字ならば何でも可

……拡張子が ".OBJ" と ".EXE" はさしあたり必要がないので削除する

%は1つだけとなり普通にキーボードからFORコマンドを実行するときと同じ形になる

%fに ".OBJ","．EXE"が順番に代入される

バッチ処理によって自動的に出力される(ECHO ONの場合)

```
FUNC-0B  COM       123  84-02-25   8:21
      14 個のファイルがあります
   746496 バイトが使用可能です

A>for %d in (00 06 0b) do type func-%d.asm □
```

……キーボードからFORコマンドを実行する"FUNC-00.ASM"と"FUNC-06.ASM" "FUNC-0B.ASM" を順にタイプアウトする

```
A>type func-00.asm ……FORコマンドが自動的に出力する
        page    255,80

;-----------------------------------------;
;   Function  00h: Terminate Program      ;
;             01h: Read keyboard and Echo ;
;             02h: Display Character      ;
;-----------------------------------------;
;
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     100h

cseg    ends
;
        end     start

A>type func-06.asm
        page    255,80

;-------------------------------------;
;   Function  06h: Direct Console I/O ;
;-------------------------------------;
;
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     100h

start:  mov     ah,6
        mov     dl,0ffh

return: int
;
cseg    ends
;
        end     start

A>type func-0b.asm
        page    255,80

;-----------------------------------------;
;   Function  0bh: Check Keyboard Status  ;
;             2ah: Get Date               ;
;             2bh: Set Date               ;
;             2ch: Get Time               ;
;             2dh: Set Time               ;
;-----------------------------------------;
;
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     100h
;
convert:aam
        or      ax,3030h
        mov     byte ptr [bx],ah
        mov     byte ptr [bx]+1,al
        ret
;
month   db  31,28,31,30,31,30,31,31,30,31,30,31
time    db  '00:00:00',0dh,'$'
;
cseg    ends
;
        end     start

A>
A>■
```

GOTO

内部2.0

機能　バッチ処理の流れを〈ラベル〉で定義されている位置に移します．
書式　GOTO〈ラベル〉

解　説

GOTOコマンドはBASIC，FORTRANなどのGOTO文と同様，〈ラベル〉の定義されている行以降へその制御を移すことができるので，IFコマンドと組み合せることによって〈条件〉による分岐，バッチ処理のループを作ることができます．

〈ラベル〉はバッチファイル中のコロン〝：〞ではじまる行で定義され，コロンに続く8文字がラベル名として識別されます．この行はラベルとして働くほかは，処理の際には無視されますので，コメント行として使うこともできます．

サンプル・オペレーション

パラメータとして与えられたファイル名で指定されるファイルをタイプアウトするバッチファイルを作成します．
(参考：IF，SHIFT)

```
A>edlin typefile.bat ……パラメータとして指定されたすべてのファイルをタイプアウトするバッチ
新しいファイルです          ファイル（ただしワイルドカードは使えない）をEDLINで作る
*i ……インサートコマンド
     1:*rem *** 連続 t y p e ***
     2:*:loop
     3:*if "%1"=="" goto endbat ┐
     4:*type %1                 │ パラメータとして与えられたファイルをすべてタイプアウトしたとき
     5:*shift                   │
     6:*goto loop               │
     7:*:endbat ←┘
     8:*^Z
*e ……EDLINから抜けDOSのコマンドレベルへ戻る

A>chdir cal

A>dir fact

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥cal

FACT     PAS       278  84-02-23   1:51
FACT     C         219  84-02-25   9:51
       2 個のファイルがあります
   741376 バイトが使用可能です

A>typefile fact.pas fact.c ……"FACT．PAS","FACT．C" をタイプアウトする

A>rem *** 連続 t y p e *** ……REM（注釈）

A>if "fact.pas"=="" goto endbat ……%1に"FACT．PAS"が割り当てられているからIFコマンドの条件は成り立たない
```

```
A>type fact.pas
program factorial(input,output);

  var n : integer;

  function fact(n : integer) : integer;
    begin
      if n=0 then fact:=1
             else fact:=n*fact(n-1);
    end;

begin
  writeln;
  write('n = ? ');
  readln(n);
  writeln(n:5,' != ',fact(n));
end.
```

"FACT．PAS" をタイプアウトする

```
A>shift
```

……パラメータをシフト．%1に次の "FACT．C" が割り当てられる

```
A>goto loop
```

……loopへ飛ぶ

```
A>if "fact.c"=="" goto endbat

A>type fact.c
main(argc,argv)

int     argc;
char    *argv[];
{
        int n;
        if (argc==1) return;
        sscanf(argv[1],"%d",&n);
        printf("%d!=%d¥r¥n",n,fact(n));
}

fact(n)
int     n;
{
        if (n<=1) return 1;
        else return n*fact(n-1);
}
```

"FACT．C" をタイプアウトする

```
A>shift
```

……パラメータをシフト．パラメータはもうないから%1は空になる

```
A>goto loop

A>if ""=="" goto endbat
```

……条件が成立して endbat へ飛びバッチ処理を終わる

```
A>
A>■
```

IF

内部2．0

機能　条件判断に応じてバッチ処理を行います．

書式　IF ［NOT］〈条件〉〈コマンド〉

解　説

IF コマンドは，〈条件〉に応じてバッチ処理を行うためのコマンドで，主に〈コマンド〉に GOTO コマンドを用いてバッチ処理の分岐に使われます．

〈条件〉には次の3種類があり，そのいずれかを指定します．

ERRORLEVEL〈数値〉

直前に実行したコマンド(プログラム)の終了コードが〈数値〉以上のときに真となり〈コマンド〉が実行されます．ERRORLEVELの設定に関するコマンドはありませんが，ユーザーのプログラムによってエラーコード(終了コード)を設定し，それをIFコマンドによって判断することができます．

〈文字列1〉＝＝〈文字列2〉

パラメータの置換え(パラメータ%0～%9に値を割り当てる．〝%〈環境変数〉%〟を，SETコマンドなどでセットされた文字列に置き換える)が終了した後，〈文字列1〉と〈文字列2〉が等しいときに真となり，〈コマンド〉が実行されます．このとき，文字列中に区切り記号(スペース，カンマ，セミコロン，等号，タブ)が含まれていてはいけません．

この指定を用いると，バッチ処理のパラメータが，ある特定の文字列と等しいかどうかを判定したり，パラメータをSHIFTコマンドでシフト後に，まだ対応する値が残っているか否かを判定することができます．

EXIST〈ファイル名〉

〈ファイル名〉で指定されるファイルが存在するときに真となり〈コマンド〉が実行されます．

これらの条件の前にNOTを付けると，条件を逆にすることができ，〈条件〉が偽である場合〈コマンド〉が実行されます．

なお，IFコマンドに続く〈コマンド〉はただ1つしか許されていませんので，それ以上の数のコマンドを実行したい場合はGOTOコマンドを用いて分岐することになります．

(参考：システムコール　4CH，4DH，GOTO)

サンプル・オペレーション

バッチファイルを用いたLOGINコマンドを示します．このLOGINコマンドは，あらかじめ作っておいたユーザーのディレクトリに移って，PATHなどの環境変数を再設定し，そのユーザーのディレクトリに〝AUTOEXEC．BAT〟が存在した場合それを実行するというものです．LOGINコマンド中で，パラメータの有無の判定および〝AUTOEXEC．BAT〟の有無の判定にIFコマンドを使用しています．ただ，IF EXISTではディレクトリがあるかないかを知ることができないので，存在しないユーザーディレクトリにログインしようとするとエラーになります．

```
A>type login.bat↵……バッチ処理を用いたLOGINコマンド．〝LOGINユーザー名〟を入力するとそのユーザーの
rem  login command   ディレクトリにはいることができる
  echo off
  if "%1"=="" goto error……ユーザー名が入力されなかったら"error"へ飛びこのコマンドの使用法を表示する
  cls
  echo ***  Welcome to PC-9801 MS-DOS, Mr.%1  ***
                                          └ユーザー名がはいる
  echo
  set user=%1……環境変数〝USER〟にユーザー名をセットする
```

```
  set home=¥usr¥%user% ……"HOME"(ユーザーのホームディレクトリ)をセットする
  chdir %home% ……ユーザーのディレクトリへ移る
  path a:¥;a:¥com;a:¥bin ……パスを設定する
  prompt $n$g
  if exist autoexec.bat autoexec ……ユーザーのホーム・ディレクトリに"AUTOEXEC.BAT"が存在すれば,それを実行する
  goto end ……"AUTOEXEC.BAT"がなければENDへ飛びバッチ処理を終わる  (ただし,バッチファイル中で,バッチ処理のネスティングができないので"AUTOEXEC.BAT"を実行してそのままバッチ処理を終わってしまう)
rem  login error
:error
  echo ***  Usage : login 'user name'  *** ……エラーメッセージ
  echo ……少なくとも2つスペースが必要,そうしないとECHO↵としたときと同じになり,そのときのECHOのON/OFFを表示してしまう
:end

A>dir u*↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

USR          <DIR>      84-04-02  15:19 ……サブ・ディレクトリ"USR"この下に各ユーザーのディレクトリがある
       1 個のファイルがあります
   436224 バイトが使用可能です

A>dir usr /w↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥usr

.          ..          NOBUT       CHI         KAWA
UCHI       KOMA
       7 個のファイルがあります
   436224 バイトが使用可能です
                                  ……各ユーザーのディレクトリ.このディレクトリ名をLOGINコマンドのパラメータとする

A>login nobut↵……いよいよLOGIN コマンドを試す

A>rem  login command

A>  echo off
***  Welcome to PC-9801 MS-DOS, Mr.nobut  *** ……画面がクリアされて,メッセージが表示される

A:dir /w↵……LOGIN コマンドから"AUTOEXEC.BAT"が実行されてプロンプトが変更されている

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥usr¥nobut
                  ……ユーザー"NOBUT"のディレクトリに移っている

.          ..          AUTOEXEC BAT  BIN
       4 個のファイルがあります
   436224 バイトが使用可能です

A:type autoexec.bat↵
prompt $n:……プロンプトの設定
path %path%;a:%home%¥bin ……パスの再設定.これにより¥USR¥NOBUT¥BINからも各コマンドを捜すようになる
                  ……それまで設定されているパスがはいる.この場合A:¥;A:¥COM;A:¥BIN
A:■
```

PAUSE

内部 1 .25

機能　バッチ処理を一時停止します.

書式　PAUSE [〈コメント〉]

解　説

PAUSE コマンドは，バッチ処理実行中にディスケットを交換したりほかのことを行う必要があるときに，バッチ処理を一時停止するためのものです．バッチ処理が PAUSE コマンドに達すると，実行を停止し，CTRL-C 以外のキーの入力で処理が再開されます．ここで CTRL-C が入力された場合

バッチ処理を中止しますか〈Y／N〉？

と聞きますから，〝Y〟または〝N〟で答えます．これに〝Y〟を入力すると，バッチ処理はそこで打ち切られ，通常のコマンド待ちの状態に戻ります．

処理が停止したときに，ユーザーに指示を与える場合，PAUSEコマンドに続けてコメントを書きます．ただしECHO OFFの状態でPAUSEコマンドが実行されると，PAUSEコマンド自身も〈コメント〉もスクリーンに表示されません．

(参考：ECHO)

サンプル・オペレーション

ディスケットを入れ換えながら，それぞれのディスクに対してCHKDSKコマンドを実行するバッチファイルを作成します．ディスケットを入れ換えるために，PAUSEコマンドによってバッチ処理を一時停止してメッセージを表示します．このとき，CTRL-C以外のキーを押すとディスクのチェックがはじまり，CTRL-Cを押すと，バッチ処理から抜けることができます．

```
A>type chk.bat↵……ディスケットを入れ換えながら，それぞれCHKDSKコマンドを実行するバッチファイル
:loop
echo off
echo ディスクの状態をチェックします }メッセージ
echo on
pause ドライブ B：にディスケットをセットしてください……バッチ処理を一時停止してメッセージを表示する．このときにディスケットの入れ換えを行う（ECHO OFFの状態だとPAUSEコマンド自体表示されないのでECHOをONにしている）
echo off
chkdsk b:
goto :loop

A>chk.↵……バッチ処理の実行

A>echo off
ディスクの状態をチェックします……ECHOコマンドによるメッセージ

A>pause ドライブ B：にディスケットをセットしてください
準備ができたらどれかキーを押してください.↵……ドライブBにチェックしたいディスケットを入れ，何かキーを押す
  └自動的に出力される
A>echo off
ディスク MS-DOS HB   は 1984-02-22 15:16 に作成されました

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     9216 バイト : 9 個のディレクトリ
   440320 バイト : 75 個のユーザーファイル
   741376 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   201904 バイト : 使用可能メモリ

ディスクの状態をチェックします

A>pause ドライブ B：にディスケットをセットしてください
準備ができたらどれかキーを押してください.

A>echo off
ディスク FUNC CALL   は 1984-02-24 1:50 に作成されました

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     8192 バイト : 7 個のディレクトリ
   681984 バイト : 162 個のユーザーファイル
   500736 バイト : 使用可能ディスク容量
```

```
   262144 バイト : 全メモリ
   201904 バイト : 使用可能メモリ

ディスクの状態をチェックします

A>pause ドライブ B:にディスケットをセットしてください
準備ができたらどれかキーを押してください^C ……STOPまたはCTRL-Cを入力すると，バッチ処理を中止するかを聞
                                                いてくる．そのとき "y" を入力するとバッチ処理が中止される
 バッチ処理を中止しますか <Y/N> y←
A>■
```

REM

内部 1 .25

機能　バッチファイルの任意の位置にコメントを付けます．

書式　REM ［〈コメント〉］

解　説

REM コマンドを用いると，バッチファイルの任意の位置にコメントを入れることができます．ECHO ON の状態でバッチファイルを実行すると，REM を含めたコメントがスクリーンに表示されますので，REM コマンドでユーザーにメッセージを与えることもできます．

(参考：ECHO)

サンプル・オペレーション

Cのソース・プログラムを，Lattice C Compiler を用いてコンパイルするためのバッチファイルを示します．この実行例では REM コマンドを用いて，バッチファイル中にコメントを入れています．なお，REM コマンドはバッチファイルの実行時に，ECHO ON の場合スクリーンに表示されますが，ECHO OFF の場合にはスクリーンに表示されないことに注意してください．この例では，後ろ3行の REM コマンドがその直前にある ECHO OFF コマンドのために，表示されていません．

```
A>type cc.bat……Cのソース・プログラムをコンパイルするバッチファイル
rem
rem Lattice C compiler      } 注釈
rem
rem one pass
b:lc1 %1 -ib:¥……コンパイラone pass
rem two pass
b:lc2 %1        ……コンパイラtwo pass      } コンパイラはドライブBに置くことにしてある
rem link
b:link b:cs %1,%1,nul,b:lcs ……ライブラリとリンクする
echo off
rem
rem End of compile
rem

A>chdir cal

A>type ackerf.c……(Ackermann関数)
#include <stdio.h>

main(argc,argv)
```

```
int     argc;
char    *argv[];
{
        int m,n;
        if (argc<=2) return;
        sscanf(argv[1],"%d",&m);
        sscanf(argv[2],"%d",&n);
        printf("A(%d,%d)=%d¥n",m,n,a(m,n));
}

a(m,n)
int     m,n;
{
        if (m==0) return ++n;
        else if (n==0) return a(m-1,1);
        else return a(m-1,a(m,n-1));
}
```

```
A>cc ackerf↵ ……"ACKERF．C" をコンパイルする

A>rem

A>rem Lattice C compiler

A>rem

A>rem one pass

A>b:lc1 ackerf -ib:¥
Lattice C Compiler (Phase 1) V2.00
Copyright (C) 1982 Lattice, Inc.

A>rem two pass

A>b:lc2 ackerf
Lattice 8086 Code Generator V2.00
Copyright (C) 1982 Lattice, Inc.
Module size P=00BC D=0010

A>rem link

A>b:link b:cs ackerf,ackerf,nul,b:lcs

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

A>echo off
A>dir ackerf↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥cal

ACKERF   C        315  84-04-03  15:19
ACKERF   EXE    13528  84-04-03  15:28
ACKERF   OBJ      473  84-04-03  15:27
       3 個のファイルがあります
  397312 バイトが使用可能です

A>ackerf 0 0 ……コンパイルしたプログラムを実行する
A(0,0)=1 ……A(0, N)= n+1

A>ackerf 1 1↵
A(1,1)=3 ……A(1, n)= 2+(n+3)-3

A>ackerf 2 2↵
A(2,2)=7 ……A(2, n)= 2(n+3)-3

A>ackerf 3 3↵ ……A(3,n)= 2^(n+3) - 3
A(3,3)=61                  (n+3 個の 2)

A>ackerf 4 4↵          } A(4, n)= 2^2^2^2^2 - 3
*** STACK OVERFLOW *** } スタックがオーバーフローしてしまった（答えは 2^2^2^2^2 - 3 となる）

A>■
```

ECHO ONの状態なのでREMコマンドを含めたコマンドが表示されている

one pass

two pass

ライブラリのリンク

ECHOをOFFにするとREMコマンドを含めたコマンドは表示されなくなる
ただしバッチ処理が終了すると自動的にECHO ONになる

ACKERF EXE …リンカでライブラリとリンクした実行ファイル

ACKERF OBJ …コンパイラが生成したオブジェクトファイル

SHIFT

内部2.0

機能　バッチファイル中のパラメータ(%0～%9)と実行時に指定される値との対応を1つずつずらします.

書式　SHIFT

解　説

SHIFTコマンドは，バッチファイル中のパラメータ(%0～%9)と実行時に指定された値との対応を1つずつずらします．具体的には，それまで%1に割り当てられていた値を%0に%2を%1にといった具合に1つずつシフトします．%9にはそれまであてはめられていなかった値のうち，最初のものが割り当てられ，%0に割り当てられていた値は失われます．なお，パラメータに割り当てる値がないときには，パラメータには空文字が割り当てられます．

またSHIFTコマンドを用いると，10個以上の値を使用できるようになります．

(参考：IF GOTO)

サンプル・オペレーション

SHIFTコマンドの例としてSHIFTで値がなくなるまで%0～%9をスクリーンに表示するバッチファイルを示します．

```
A>type test.bat↵
echo off
rem
rem  sample for shift command
rem
:loop
   if "%0"=="" goto end……パラメータがなくなったら終わり
   echo %0 %1 %2 %3 %4 %5 %6 %7 %8 %9……%0～%9までに対応する値を表示する
   shift……パラメータをシフトする
   goto loop
:end

A>test 0 1 2 3 4 5 6 7 8 9 a b c d e f↵

A>echo off
test 0 1 2 3 4 5 6 7 8 ……最初%0にはバッチファイルの名前(拡張子を除く)"TEST"がはいっている
0 1 2 3 4 5 6 7 8 9 ……一度シフトすると今まで%0～9に割り当てられていなかった"9"が%9に割り当てられる
1 2 3 4 5 6 7 8 9 a
2 3 4 5 6 7 8 9 a b
3 4 5 6 7 8 9 a b c
4 5 6 7 8 9 a b c d
5 6 7 8 9 a b c d e
6 7 8 9 a b c d e f
7 8 9 a b c d e f ←――――%9に割り当てるパラメータがなくなったので空文字が割り当てられる
8 9 a b c d e f
9 a b c d e f
a b c d e f
b c d e f
c d e f
d e f
e f
f

A>■
```

2章 EDLIN

EDLIN の起動

EDLIN を起動するには次の書式に従って入力を行います.

A > EDLIN 〈ファイル名〉 [／B] ↵

〈ファイル名〉　：これから作成／編集しようとするファイルの名前を指定します.〈ファイル名〉がディレクトリに登録されていない場合には，新規ファイルとしてディレクトリに登録し，次のようなメッセージとプロンプト〝*〟を表示してコマンドの入力待ちになります.

新しいファイルです
*

／B スイッチ　：通常，EOF(ファイルの終わり)(1AH)が見い出されたところでファイルの読込みを終了しますが，／B スイッチを付けると，ファイルの途中にある EOF もテキストとして読み込むので，途中に EOF のはいったファイルを編集することができます. ただし，ファイルの最後の EOF は，／B スイッチを付けてもテキストとして読み込まれません.

コマンドの入力方法

(1)行番号は現在行(行番号の隣に〝*〟の表示された行)からの相対位置で指定できます. 現在行より前をマイナス〝−〟，後ろをプラス〝+〟で表します.

［例］ −10, +10L （現在行とその前後 10 行の内容を表示する）

(2)複数のコマンドを，〝;〟で区切って 1 行にまとめて書くことができます.

［例］ 15；−5, +5L （第 15 行を編集して，その行と前後 5 行を表示する）

(3)行番号とコマンドの間にはスペースを入れても入れなくても，どちらでもかまいません.

(4)コントロール・キャラクタコードを入力するには，まず CTRL-V を入力し，次にそれに対応するアルファベットを大文字で入力します.

［例］ CTRL-C をテキスト中に入れる場合

……CTRL−V C……

(5) MS-DOS のコントロール・キャラクタとテンプレート機能は，そのまま EDLIN 内でも使用できます．入力は大文字でも小文字でもかまいません．

パラメータ

EDLIN のいくつかのコマンドは，パラメータとともに用いますが，以下にそのパラメータの書式について解説します．

〈行番号〉　〈行番号〉はコマンドの働く行やその範囲を指定します．〈行番号〉は〝,〟またはスペースによって他の行番号，他のパラメータから分離されていなければなりません．

〈行番号〉の指定には以下の 3 つの方法があります．

(1)数字……65529 以下の整数で，直接行番号を指定します．
現在の最大の行番号より大きな数を指定すると，〈行番号〉は自動的に最後の行番号の次の行(〝#〟の指定と同じ)になります．

(2)ピリオド〝.〟…行番号としてピリオドを用いると，現在行(行番号の隣に〝*〟が表示されている行)を指定することができます．

(3)〝#〟……行番号としてシャープを用いると，最後の行の次の行番号を指定することができます．

なお，上に挙げた方法で〈行番号〉の指定を行わずに省略すると，そのコマンドにおけるデフォルト値が用いられます．

疑問符〝?〟　疑問符は，Search コマンド，Replace コマンドで使用します．疑問符をそれらのコマンドとともに用いると，EDLIN は捜し出した文字列が目的のものであるかどうかをたずねるメッセージを出力し，入力待ちになりますから，ユーザーは〝Y〟または〝N〟で答えます．

〈文字列〉　〈文字列〉は，Search コマンド，Replace コマンドで使用し，検索する文字列や置換する文字列を指定します．スペースやカンマを〈文字列〉の中に入れることができるので，〈文字列〉を区切るには CTRL-Z を用います．

Append

機能　ディスク上のファイルから，メモリ上のファイルへ〈n〉行読み込んで追加します．
書式　［〈n〉］A

解　説

Appendコマンドは Writeコマンドと併せて，メモリにはいりきらないファイルを編集する際に用います．編集しようとするファイルがメモリにはいりきらない場合には，それを少しずつメモリにはいる分だけ読み込んできてそれについて編集を行い，編集の終わった部分をディスクに書き込んだ後に，はいりきらなかった分を読み込んできて編集を行います．

メモリにはいりきらないファイルを指定して EDLIN を起動すると，EDLIN はメモリの空き領域の4分の3に達するまでファイルを読み込み，プロンプト〝*〟を表示してコマンドの入力待ちになります．ユーザーは，残り4分の1のメモリを使ってメモリに読み込んだファイルの編集を行うことができます．

メモリにはいりきらなかった部分を編集するには，まず編集の終わった部分を Writeコマンドで先頭からファイルに書き込み，メモリに空き領域を確保します．次に Appendコマンドを使って，ディスク上のファイルの残りを読み込んで編集を行います．

Appendコマンドは指定した行数の途中でファイルの終わりに達すると，

ファイルを終りまで読込みました

というメッセージを表示します．

なお，行番号はメモリ中にあるファイルに対してのみ1から順に付けられます．

(参考：Write)

① 〈n〉 A

現在メモリにあるファイルへ〈n〉行追加します．

```
A>dir
 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥edwork

.            <DIR>      84-03-25   3:33
..           <DIR>      84-03-25   3:33
MEMDUMP  ASM     52487  84-03-25   3:27
       3 個のファイルがあります
   661504 バイトが使用可能です
```

```
A>edlin memdump.asm ……"MEMDUMP.ASM" を編集するためEDLINを起動する
*90l ……メモリにはいりきらなかったので "ファイルを終りまで読込みました" というメッセージは表示されない
    90:          cmp     al,'z'
    91:          ja      notlow
    92:          sub     al,'a'-'A'
    93: notlow:  ret
    94: ;
    95: cseg     ends
    96: ;
    97:
    98: ;-----------------------------------;
    99: ;        Dump CX bytes from DS:BX     ;
   100: ;-----------------------------------;
   101: ;
   102: cseg     segment
   103:          assume  cs:cseg,ds:cseg
                 public  dump,hexdisp,crlfout,putch
                 near
   110:
   111:          and
   112:          shr     cx,1
*98 ……第98行を編集する
    98:*;-----------------------------------;
    98:*;***********************************;
*100
   100:*;-----------------------------------;
   100:*;***********************************;
*p ……次のページをスクリーンに表示する
   101: ;
   102: cseg     segment
   103:          assume  cs:cseg,ds:cseg
                 public  dump,hexdisp,crlfout,putch
                 near                                  } メモリ中のファイルの編集完了
   120: hex
   121:          call
   122:          mov     al,
   123:*         call    putch
*1000w ……編集の終わったメモリ中のファイルを先頭から1000行出力ファイルに書き込む
*1000a ……メモリにはいりきらなかった分を入力ファイルから1000行読込みメモリ中のファイルの後ろに追加する
ファイルを終りまで読込みました ……読込みの途中でファイルが終わりに達したので，そのメッセージが出力されている
*l
     1:*         call    putch
     2:          pop     bx
     3:          mov     cx,10h
     4: chrlp:   mov     al,[bx]
                 cmp     al,' '
                 jb      tabchr
    21:          pop
    22:          pop     ax
    23:          ret
*20
    20:*next:    pop     cx
    20:*next:    pop     cx      ;repair cx register
*e ……ファイルをセーブしてEDLINを終了する

A>■
```

② A

〈n〉を省略するとメモリの 4 分の 3 に達するまでファイルを読み込みますが，すでにメモリの 4 分の 3 が使用されている場合には何も起こりません．

```
A>dir direc.c

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥edwork

DIREC    C       53745  84-03-25   3:46
        1 個のファイルがあります
   553984 バイトが使用可能です
```

```
A>edlin direc.c.  ……EDLINで"DIREC.C"を編集する．メモリにはいりきらなかったので"ファイルを終りまで読込みました"というメッセージは表示されない
*l
    1:*#include <stdio.h>
    2: #include <ctype.h>
    3: #include <dos.h>
    4:
    5: #define FALSE 0
    6: #define TRUE  1
    7: #define MAXLEVEL 4
    8:
    9: char    filename[13];
   10: char    line[81];
   11: char    nline[81];
   20: {
   21:     int
   22:
   23:     if (argc > 3)
*9i.  ……第9行の前にテキストをインサートする
    9:* /* define static variables */
   10:*
   11:*^Z.  ……CTRL-Z.を入力してインサートモードを抜ける

*1l
    1: #include <stdio.h>
    2: #include <ctype.h>
    3: #include <dos.h>
    4:
    5: #define FALSE 0
    6: #define TRUE  1
    7: #define MAXLEVEL 4
    8:
    9:  /* define static variables */
   10:
   11:*char    filename[13];
   21: char    *ar
   22: {
   23:     int     level, nmptr;
*w.  ……メモリ中のファイルの4分の3を出力ファイルに書込む
*a.  ……メモリにはいりきらなかった分をメモリの4分の3がいっぱいになるまで読込む
ファイルを終りまで読込みました  ……読込みの途中でファイルが終わりに達した
*1000l
 1000:                 nmptr--;
 1001:             }
 1002:             else
 1003:                 ndisp = FALSE;
                   if (exist)
 1019:                 (level, nmptr);
 1020:     if (!search
 1021:     {
 1022:         if (!ndisp)
*1013i
 1013:* /* search files */
 1014:*
 1015:*^Z
*l
 1004:             if (exist)
 1005:             {
 1006:                 find (level, nmptr);
 1007:             }
 1008:         }
           if (!exist) printf ("Sorry not found¥n");
 1022:     if (
 1023:     {
 1024:         if (!ndisp)
 1025:         {
 1026:             strcpy (filename, &dta[30]);
*e.  ……ファイルをセーブしてEDLINを抜けDOSのコマンドレベルに戻る

A>■
```

Copy

機能　〈行番号 1〉，〈行番号 2〉で指定された範囲の行を〈行番号 3〉の直前に，〈回数〉で指定されるだけコピーします．

書式　[〈行番号 1〉]，[〈行番号 2〉]，〈行番号 3〉[，〈回数〉] C

解　説

Copy コマンドは，〈行番号 1〉と〈行番号 2〉の範囲の行を〈行番号 3〉の直前に〈回数〉だけコピーします．テキスト中に同じあるいは似た内容の行が複数ある場合には，編集作業が容易に行えます．

〈行番号 1〉，〈行番号 2〉は省略できますが，省略すると現在行(行番号の隣に〝*〟が表示されている行)が入力されたとみなされ，また〈回数〉を省略すると 1 が指定されたとみなされ，コピーは 1 回だけ行われます．

```
*l
    1:*cseg     segment
    2:          assume  cs:cseg
    3: ;
    4:          org     100h
    5: ;
    6: loop:    mov     ah,3
    7:          int     21h
    8:          cmp     al,1ah      } この範囲を第14行の前にコピーする
    9:          je      return      }
   10:          mov     ah,2
   11:          mov     dl,al
   12:          int     21h
   13:          jmp     loop
   14: ;
   15: return:  int     20h
   16:
   17: cseg     ends
   18:
   19:          end
*8,9,14c ……第8～第9行を第14行の前にコピーする
*l
    3: ;
    4:          org     100h
    5: ;
    6: loop:    mov     ah,3
    7:          int     21h
    8:          cmp     al,1ah      } コピーもとのテキストも残っている
    9:          je      return      }
   10:          mov     ah,2
   11:          mov     dl,al
   12:          int     21h
   13:          jmp     loop
   14:*         cmp     al,1ah      } コピーされている
   15:          je      return      }
   16: ;
   17: return:  int     20h
   18:
   19: cseg     ends
   20:
   21:          end
```

```
*16,17,20,3c↵ ……第16～第17行を第20行の前に3回コピーする
*1↵
    9:          je      return
   10:          mov     ah,2
   11:          mov     dl,al
   12:          int     21h
   13:          jmp     loop
   14:          cmp     al,1ah
   15:          je      return
   16: ;                                  } コピーもとの内容
   17: return:  int     20h
   18:
   19: cseg     ends
   20:*;
   21: return:  int     20h
   22: ;
   23: return:  int     20h                } コピーされている
   24: ;
   25: return:  int     20h
   26:
   27:          end
*■
```

なお，ファイルの1行1行にEDLINが編集用に付けた行番号は自動的に付け換わります．また〈行番号1〉，〈行番号2〉で指定される範囲と〈行番号3〉が重複してはいけません．〈行番号3〉は〈行番号1〉以下の数，あるいは〈行番号2〉より大きな数で指定します．

(参考：Moveコマンド)

Delete

機能　指定された行，あるいは指定された範囲の行を削除します．

書式　[〈行番号1〉][，〈行番号2〉]D

解　説

Deleteコマンドは，〈行番号1〉で指定された行，あるいは〈行番号1〉と〈行番号2〉で指定された範囲の行を削除します．

〈行番号1〉を省略すると現在行が指定されたとみなされ，また〈行番号2〉を省略すると，〈行番号1〉で指定された行のみが削除されます．現在行は削除された範囲のすぐ次の行に移り，行番号が付け直されます．

```
*1l↵
   1:*3.5  FILE ALLOCATION TABLE (FAT)
   2:
   3:      The File Allocation Table  is  an array of
   4: 12-bit entries (1.5 bytes) for each cluster  on
   5: the  disk.   The first two FAT  entries  map  a
   6: portion  of  the directory; these  FAT  entries
   7: indicate the size and format of the disk.
   8:
   9:      The second and third bytes currently
                 always contain FFH.                  } この範囲の行を削除する

                                    which starts at byte
```

```
      22: clusters made
      23: allocated for new files.
*8,11d.....第8～第11行を削除する
*11
       1: 3.5  FILE ALLOCATION TABLE (FAT)
       2:                                                    }この行を削除する
       3:      The File Allocation Table  is  an array of
       4: 12-bit entries (1.5 bytes) for each cluster  on
       5: the  disk.   The first two FAT  entries  map  a
       6: portion  of  the directory; these  FAT  entries
       7: indicate the size and format of the disk.
       8:*     The third FAT entry,  which starts at byte.....削除されている
       9: offset 4,  begins  the mapping of the data area
      10: (cluster 002).  Files in the data area  are not
          written  sequentially  on the disk.  The
          located one cluster  at a time,
      18: clusters ma
      19: allocated for new f
*2d.....第2行のみを削除する
*11
       1: 3.5  FILE ALLOCATION TABLE (FAT)
       2:*     The File Allocation Table  is  an array of.....削除されている
       3: 12-bit entries (1.5 bytes) for each cluster  on
       4: the  disk.   The first two FAT  entries  map  a
       5: portion  of  the directory; these  FAT  entries
       6: indicate the size and format of the disk.
          The third FAT entry,  which starts at byte
          ins  the mapping of the data area
      15: location
      16: efficient  utiliza  in the data area  are not
      17: clusters made available by
      18: allocated for new files.
*,#d.....現在行(第2行)からファイルの終わりまでを削除する
*11
       1: 3.5  FILE ALLOCATION TABLE (FAT)
*■
```

edit

機能　行を指定し編集モードにはいります.

書式　[〈行番号〉]

解　説

edit コマンドは,〈行番号〉で指定された行を編集するために編集モードにはいるコマンドです.

〈行番号〉を入力すると，EDLIN はその行の行番号とテキストを表示し，その下に再び行番号を表示して入力待ちになります(編集モードにはいる).

〈行番号〉を省略してリターンキーのみを入力すると，現在行(行番号の隣に〝*〟が表示されている行)の次の行が入力されたとみなされます.

編集モードで新しいテキストを入力すると，その行の内容が新しく入力されたテキストに変更されます．編集を行う前のテキスト(上の行に表示されているテキスト)はテンプレートにコピーされているので，このテンプレート機能を使って編集を行うことができます．たとえば，その行のテキストの一部のみを変更したい場合などには便利でしょう.

編集モードにおいて，その行を変更する必要がない場合には，何も入力しないでそのままリターンキーを押します．また，何か入力した後でも CTRL-C を押すとその行は変更されません.

```
*l
   1:*main(argc,argv)
   2:  .
   3: int      argc;
   4: char     *argv[];
   5: {
   6:          int m,n;
   7:          if (argc<=2) return;
   8:          sscanf(argv[1],"%d",&m);
               sscanf(argv[2],"%d",&n);
  14: int      m,
  15: {
  16:          if (m==0) return
  17:          else if (n==0) return a(m-1,1);
  18:          else return a(m-1,a(m,n-1));
  19: }
*17,.....第17行を編集する
  17:*          else if (n==0) return a(m-1,1);……EDLINから自動的に出力される
  17:*             else if (n==0) return a(m-1,1);
*,,.....単にリターンのみを入力すると次の行が編集できる
  18:*          else return a(m-1,a(m,n-1));
  18:*             else return a(m-1,a(m,n-1));
*l
   7:          if (argc<=2) return;
   8:          sscanf(argv[1],"%d",&m);
               sscanf(argv[2],"%d",&n);
               printf("A(%d,%d)=%d¥r¥n",m,n,a(m,n));
  14: int      m,
  15: {
  16:          if (m==0) return
  17:             else if (n==0) return a(m-1,1);
  18:*            else return a(m-1,a(m,n-1));
  19: }
*■
```

End

機能　ファイルをディスクにセーブし，EDLIN を終了します．

書式　E

解　説

End コマンドは，編集の終わったファイルをディスクにセーブして EDLIN を終了し，DOS のコマンドレベルへ戻るためのものです．このとき，編集する前のファイルは拡張子が ".BAK" に付け換えられ，削除されずにバックアップとして残されます．なお，ファイルを新規に作成した場合は，バックアップ用ファイルは作られません．

```
A>dir count

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥edwork

COUNT    C          94  84-03-25   5:09
       1 個のファイルがあります
  495616 バイトが使用可能です
```

```
A>edlin count.c
ファイルを終りまで読込みました
*l
     1:*#include <stdio.h>
     2:
     3: main()
     4: {
     5:          int     i;
     6:          for (i=1;i<=100;i++){
     7:                  printf("%d ",i);
     8:          }
     9: }
*6.  ……第6行を編集する
     6:*         for (i=1;i<=100;i++){
     6:*         for (i=1;i<=100;i+=i){
*e.  ……ファイルをセーブしてEDLINを終了し,        ここを変更
       DOSのコマンドレベルに戻る
A>dir count

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥edwork

COUNT    BAK        94  84-03-25   5:09 ……バックアップ用のファイル(編集前のファイル)
COUNT    C          95  84-03-25   5:12 ……編集後のファイル
       2 個のファイルがあります
   494592 バイトが使用可能です

A>type count.c
#include <stdio.h>

main()
{
        int     i;                  変更後
        for (i=1;i<=100;i+=i){
                printf("%d ",i);
        }
}

A>type count.bak
#include <stdio.h>

main()
{
        int     i;                  変更前のまま
        for (i=1;i<=100;i++){
                printf("%d ",i);
        }
}

A>■
```

End コマンドによって編集を終了すると，つねに EDLIN 起動時に指定したドライブのディスクにセーブされます．したがって，ディスクに編集の終わったファイルを書き込む空き領域がないと，エラーが発生して書込みが中止され，そのままコマンドレベルへ戻ってしまうので，メモリ上の今編集したファイルは失われてしまいます．ですから，ユーザーは EDLIN を起動する前に，ファイル全体を書き込むだけの十分な空き領域があることを確かめておく必要があります．

なお，EDLIN を起動して End コマンドで終了するまで，その起動時に指定したファイルのあるドライブのディスケットを入れ換えてはなりません．ディスケットを入れ換えて End コマンドを実行してしまうと，ディスクの FAT やほかのファイルなどが壊れてしまう可能性がありますし，書き込んだファイルはディレクトリに登録されないので，その後使用することができなくなってしまいます．

(参考：Quit，CHKDSK／F)

Insert

機能　指定された〈行番号〉の直前にテキストを挿入します．
書式　［〈行番号〉］I

解　説

Insert コマンドは，指定された〈行番号〉(省略された場合は現在行)の直前にテキストを挿入します．またユーザーが新しくファイルを作成する場合には，Insert コマンドを用います．

Insert コマンドを入力すると，それ以降 CTRL-C または行頭で〝CTRL-Z ↵〟が入力されるまで挿入モードにはいり，リターンキーが押されるたびにそのテキストが挿入され，新しい行番号が自動的に現れます．必要な挿入が終わり挿入モードを抜けると，現在行は最後に挿入された行の次の行に移り，挿入された後に続く行番号は自動的に付け直されます．

指定した行番号が最終行の行番号より大きかったり，〝#〟を〈行番号〉として指定した場合，テキストはファイルの終わりに付け足されます．

```
A>edlin subpr.bat ……EDLINで〝SUBPR．BAT〟を作る
新しいファイルです
*i ……新しいファイルを作るときはInsertコマンドを使う
     1:*echo off
     2:*set prmt=$d$t
     3:*set prom=$n$g
     4:*prompt $e[s$e[0;0f$e[0K$e[7m%prmt%$e[0m$e[u%prom%
     5:*^Z ……CTRL-Z．を入力してインサートモードを抜ける
*4i ……第4行の前にテキストを挿入する
     4:*if not "%1"=="" set prmt=%1
     5:*if not "%2"=="" set prom=%2
     6:*^Z ……CTRL-Z．を入力してインサートモードを抜ける．この行は挿入されない
*1,#l
     1: echo off
     2: set prmt=$d$t
     3: set prom=$n$g
     4: if not "%1"=="" set prmt=%1
     5: if not "%2"=="" set prom=%2
     6:*prompt $e[s$e[0;0f$e[0K$e[7m%prmt%$e[0m$e[u%prom%
*e

A>■
```

List

機能　指定された範囲の行の内容をスクリーンに表示します．
書式　［〈行番号 1〉］［，〈行番号 2〉］L

解　説

①〈行番号 1〉,〈行番号 2〉L

〈行番号 1〉と〈行番号 2〉で指定された行の内容をスクリーンに表示します.

```
*46,571↵……第46～第57行をスクリーンに表示する
    46: hanoi(n,source,dest,work)
    47: int n,source,dest,work;
    48:
    49: {
    50:         if (n == 1) store(n,source,dest);
    51:
    52:         else    {
    53:                 hanoi (n-1,source,work,dest);
    54:                 store(n,source,dest);
    55:                 hanoi (n-1,work,dest,source);
    56:                 }
    57: }
*100,#1↵……第100行から終わりまでをスクリーンに表示する
   100:
   101: locate(x,y)
   102: int x,y;
   103:
   104: {
   105:         printf("¥x1b[%d;%dH",y,x);
   106: }
   107:
   108: cls()
   109: {
   110:         printf("¥x1b[2J");
   111: }
*■
```

②〈行番号 1〉L

〈行番号 1〉ではじまる 23 行の内容をスクリーンに表示します.

```
*801↵……第80行から1画面分(23行)をスクリーンに表示する
    80:                 locate((a[i]-1)*24+4,5);
    81:                 dispone(&pole[a[i]][1]);
    82:         }
    83: }
    84:
    85: dispone(pol)
    86: int *pol;
    87: {
    88:         int i,j;
    89:
    90:         for (i=0;i<max;i++){
    91:                 printf("¥x1b[s");
    92:                 for (j=0;j<max;j++){
    93:                         if (j<*(pol+i)) printf("*");
    94:                         else printf(" ");
    95:                 }
    96:                 printf("¥x1b[u");
    97:                 printf("¥x1b[1B");
    98:         }
    99: }
   100:
   101: locate(x,y)
   102: int x,y;
*■
```

③, 〈行番号 2〉 L

現在行の 11 行前から〈行番号 2〉までの内容をスクリーンに表示します．この場合〈行番号 1〉を省略したことを EDLIN に指示するために，かならずその行番号の前にカンマ〝,〟を付けなければなりません．

```
*,80↵……現在行(第70行)の11行手前から第80行までスクリーンに表示する
   59: store(n,source,dest)
   60: int n,source,dest;
   61:
   62: {
   63:         locate(15,23);
   64:         printf("No. %d Disk from %d pole to %d pole ",n,source,dest);
   65:         pole[dest][pp[dest]--]=n;
   66:         pole[source][++pp[source]]=0;
   67:         disp(source,dest);
   68: }
   69:
   70:*disp(source,dest)……現在行(行番号の次に〝*〟が付いている)
   71: int     source,dest;
   72:
   73: {
   74:
   75:         int a[3];
   76:         int i;
   77:         a[1]=source;
   78:         a[2]=dest;
   79:         for (i=1;i<=2;i++){
   80:                 locate((a[i]-1)*24+4,5);
*■
```

④ L

行番号のどちらも省略して〝L〟のみを入力すると，現在行と前後 11 行ずつの 23 行がスクリーンに表示されます．現在行が第 11 行以下の場合，表示が 23 行になるまで，その行の後ろが表示されます．

```
*1↵……現在行(第20行)と前後11行(1画面分)をスクリーンに表示する．ただし第26行のような1行分より長い行があるとスクリーンから
                                                          はみ出てしまう
    9: main(argc,argv)
   10: int     argc;
   11: char    *argv[];
   12: {
   13:         int     n,source,dest,i,j;
   14:
   15:         if (argc == 1 || argc > 4)
   16:                 printf ("argment error");
   17:
   18:         else
   19:         {
   20:*                sscanf(argv[1],"%d",&n,);
   21:                 sscanf(argv[2],"%d",&source);
   22:                 sscanf(argv[3],"%d",&dest);
   23:
   24:                 if (n < 1 || source < 1 || dest < 1
   25:                         || source > 3 || dest > 3 || source == dest)
   26:                         printf ("Source pole or Dest pole number error")

   27:                 else if (n>max) printf("n<=%d",max);
   28:                 else {
   29:                         printf("¥x1b[>5h");
   30:                         for (i=0;i<=3;i++){
   31:                             for (j=0;j<=max;pole[i][j++]=0);
*■
```

(参考：Page)

Move

機能　指定した範囲のテキストを別の位置に移動させます.

書式　[〈行番号 1〉],[〈行番号 2〉],〈行番号 3〉 M

解　説

Move コマンドは,〈行番号 1〉,〈行番号 2〉で指定された範囲の行を〈行番号 3〉で指定された行の直前に移動します.

〈行番号 1〉,〈行番号 2〉は省略できますが,省略すると現在行が入力されたものとみなされます.

なお〈行番号 1〉,〈行番号 2〉で指定される範囲と,〈行番号 3〉が重複してはなりません.具体的には,〈行番号 3〉は〈行番号 1〉以下の数,あるいは〈行番号 2〉より大きな数を指定します.

```
*1
    1:*cseg     segment
    2:          assume  cs:cseg
    3: ;
    4:          org     100h
    5: ;
    6: loop:    mov     ah,3
    7:          int     21h
    8:          cmp     al,1ah
    9:          je      return
   10:          mov     ah,2
   11:          mov     dl,al
   12:          int     21h
   13:          jmp     loop
   14: ;
   15: return:  int     20h
   16:
   17: cseg     ends
   18:
   19:          end
*6,9,14m
*1l
    1: cseg     segment
    2:          assume  cs:cseg
    3: ;
    4:          org     100h
    5: ;
    6:          mov     ah,2
    7:          mov     dl,al
    8:          int     21h
    9:          jmp     loop
   10: loop:    mov     ah,3
   11:*         int     21h
   12:          cmp     al,1ah
   13:          je      return
   14: ;
   15: return:  int     20h
   16:
   17: cseg     ends
   18:
   19:          end
*■
```

〈行番号 1〉,〈行番号 2〉で指定された範囲の行を〈行番号 3〉で指定された行の直前に移動する

……COPY コマンドと違って移動もとの行は削除される

移動された

(参考：Copy)

Page

機能　テキストを23行ずつページに分けて表示します.

書式　[〈行番号1〉][,〈行番号2〉]P

解　説

Pageコマンドは,〈行番号1〉で指定された行から〈行番号2〉で指定された行までをスクリーンに表示します.〈行番号2〉を省略して用いると,〈行番号1〉ではじまる23行を表示して,現在行を,表示された最後の行に移します.基本的にはListコマンドと似ていますが,現在行の変更および〈行番号2〉を省略した場合の動作が異なっています.

```
*1,5p……第1～第5行までスクリーンに表示する現在行が第5行に移ることがListコマンドと異なる
     1: 3.5  FILE ALLOCATION TABLE (FAT)
     2:
     3:      The  following  information  is  included
     4: for  system  programmers  who  wish  to  write
     5:*installable  device  drivers.  This  section……現在行が移動している
*p……現在行(第5行)の次から1画面分スクリーンに表示する. 現在行はスクリーンに表示されたいちばん下の行(第28行に移る)
     6: explains  how  MS-DOS uses  the File Allocation
     7: Table  to  convert  the  clusters of a file  to
     8: logical sector  numbers.  The  driver  is  then
     9: responsible  for  locating the logical sector on
    10: disk.  Programs  must  use  the  MS-DOS  file
    11: management  function calls for accessing files;
    12: programs that access the FAT are not guaranteed
    13: to be  upwardly-compatible with future releases
    14: of MS-DOS.
    15:
    16:      The File Allocation Table  is  an array of
    17: 12-bit entries (1.5 bytes) for each cluster  on
    18: the  disk.  The first two FAT  entries  map  a
    19: portion  of  the directory; these  FAT  entries
    20: indicate the size and format of the disk.
    21:
    22:      The second and third bytes currently
    23:             always contain FFH.
    24:
    25:      The third FAT entry,  which starts at byte
    26: offset 4,  begins  the mapping of the data area
    27: (cluster 002).  Files in the data area  are not
    28:*always written  sequentially  on the disk.  The
*p……次を表示する. 途中でテキストが終わっている
    29: data area is  allocated one cluster  at a time,
    30: skipping  over  clusters already allocated. The
    31: first  free  cluster  found  will  be  the next
    32: cluster  allocated,  regardless of its physical
    33: location  on the disk. This  permits the most
    34: efficient  utilization  of  disk space  because
    35: clusters made available by erasing files can be
    36:*allocated for new files.
*■
```

(参考：List)

Quit

機能　編集中のファイルをディスクにセーブせずに EDLIN を終了します.
書式　Q

解　説

Quit コマンドは,編集中のファイルをディスクにセーブせずに,EDLIN を終了してコマンドレベルに戻るためのものです.

〝Q〞を入力すると，EDLIN はメモリにあるファイルの内容を，ディスクへ書き込まなくてもよいかという確認のメッセージを表示して，入力待ちになります．そこで，〝Y〞を入力すると即座に EDLIN を終了しコマンドレベルへ戻りますが，その際メモリ中のファイルはディスクに書き込まれず，バックアップファイルも作られません．〝N〞または〝Y〞以外の文字を入力すると，編集を続けることができます.

なお，EDLIN を起動した時点ですでに作られている BAK ファイルはディスクに残っていますから，Quit コマンドで編集を中止する場合，Write コマンドでファイルの一部をディスクに書き込んでいない限りその BAK ファイルは EDLIN 起動前の状態で残っています.

```
A>dir count↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥edwork

COUNT     C          106  84-03-25   7:27
COUNT     BAK         95  84-03-25   7:26 ……すでにバックアップファイルがある
        2 個のファイルがあります
   490496 バイトが使用可能です

A>edlin count.c↵……〝COUNT．C〞をEDLINで編集
ファイルを終りまで読込みました
*l↵
     1:*/*
     2:
     3:  */
     4: #include <stdio.h>
     5:
     6: main()
     7: {
     8:          int     i;
     9:          for (i=1;i<=100;i+=i){
    10:                  printf("%d ",i);
    11:          }
    12: }
*q↵……QuitコマンドでエDLINを抜ける
編集を中止しますか <Y/N>? y …… Quitコマンドを入力すると確認を求められる．ここで〝y〞を入力するとEDLINを終了する
A>dir count↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥edwork

COUNT     C          106  84-03-25   7:27
COUNT     BAK         95  84-03-25   7:26 ……バックアップファイルもそのまま残っている
        2 個のファイルがあります
   490496 バイトが使用可能です

A>■
```

(参考：End Write)

Replace

機能　指定された範囲の行の中にある〈文字列1〉を，〈文字列2〉に置き換えます．
書式　[〈行番号1〉][,〈行番号2〉][?] R〈文字列1〉CTRL-Z [〈文字列2〉]

解　説

Replaceコマンドは，〈行番号1〉と〈行番号2〉の範囲の行の中にある〈文字列1〉を検索してそれを〈文字列2〉に置き換え，置き換えた行をそのつど表示します．

①〈行番号1〉,〈行番号2〉R〈文字列1〉CTRL-Z〈文字列2〉

〈行番号1〉と〈行番号2〉の範囲にある〈文字列1〉を，〈文字列2〉で置き換えます．
〈文字列1〉と〈文字列2〉は，CTRL-Zで区切らなければなりません．

```
*1,#↵
     1:* COMMAND.COM,  the command interpreter program
     2: which reads and interprets user keyboard input
     3: and  executes 'built-in' commands  like DIR or
     4: initiates external programs.
*1,4rprogram^ZPROGRAM↵ ……第1～4行の "program" を "PROGRAM" に置き換える
     1:* COMMAND.COM,  the command interpreter PROGRAM
     4: initiates external PROGRAMs.          } 置き換えられた行が表示される
*1,#↵
     1:  COMMAND.COM,  the command interpreter PROGRAM
     2: which reads and interprets user keyboard input
     3:*and  executes 'built-in' commands  like DIR or
     4: initiates external PROGRAMs.
*■                                  置き換えられた
```

②[〈行番号1〉][,〈行番号2〉]R〈文字列1〉CTRL-Z

〈文字列2〉を省略すると，空文字(ヌルストリング)が指定されたとみなされて，範囲内の〈文字列1〉がすべて空文字に置き換えられます．つまり，範囲内の〈文字列1〉をすべて削除するのと同じ結果が得られます．

```
*1,3rPROGRAM^Z↵ ……第1～3行の "PROGRAM" を空文字に置き換える
     1:  COMMAND.COM,  the command interpreter
*1,#↵
     1:  COMMAND.COM,  the command interpreter          ……"PROGRAM" が削除された
     2: which reads and interprets user keyboard input
     3: and  executes 'built-in' commands  like DIR or
     4: initiates external PROGRAMs.
*■                          └第4行は範囲外なので置き換えられない
```

③[,〈行番号2〉]R〈文字列1〉CTRL-Z[〈文字列2〉]

〈行番号1〉を省略すると，現在行の次の行が指定されたとみなされます．この場合，かならず〈行番号2〉の前に〝,〟を付けなければなりません．

```
*1,#l
     1:* COMMAND.COM,  the command interpreter ……現在行
     2: which reads and interpretes user keyboard input
     3: and  executes 'built-in' commands  like DIR or
     4: initiates external PROGRAMs.
*,2rin^ZOUT ……現在行の次の行(第2行)から第2行までの "in" を "OUT" に置き換える
     2: which reads and OUTterpretes user keyboard input
     2: which reads and OUTterpretes user keyboard OUTput  } 1行内で置き換えが2度起きている
*1,#l
                                  このinは範囲外なので置き換えられない
     1:* COMMAND.COM,  the command interpreter
     2: which reads and OUTterpretes user keyboard OUTput
     3: and  executes 'built-in' commands  like DIR or
     4: initiates external PROGRAMs.
*■
```

④ [〈行番号 1〉] R 〈文字列 1〉CTRL-Z [〈文字列 2〉]

〈行番号 2〉を省略すると，ファイルの最後の次の行(#)が指定されたとみなされます．

```
*2rOUT^Zout ……第2行から終わりまでの "OUT" を "out" に置き換える
     2: which reads and outterpretes user keyboard OUTput
     2: which reads and outterpretes user keyboard output
*1,#l
     1:* COMMAND.COM,  the command interpreter                    置き換えられた
     2: which reads and outterpretes user keyboard output
     3: and  executes 'built-in' commands  like DIR or
     4: initiates external PROGRAMs.
*■
```

⑤ [〈行番号 1〉] [，〈行番号 2〉]？R 〈文字列 1〉CTRL-Z [〈文字列 2〉]

R の前に "？" を与えると，Replace コマンドは〈文字列 1〉に出会うたびにその行を表示し，置き換えてもよいかを聞いてきます．このとき "Y" またはリターンキーを押すと〈文字列 1〉は〈文字列 2〉で置き換えられますが，それ以外のキーを押すと〈文字列 1〉は置き換えられずに残ります．これを指定された範囲が終わるまで繰り返すと，目的に合った〈文字列 1〉のみを置き換えることができます．

```
*1,#?rand^Z&  ……第1行から終わりまでの "and" を?オプションを付けてReplaceコマンドを実行して "&" に置き換える
     1:* COMMAND.COM,  the comm& interpreter
よろしいですか (Y/N)? n ……"n" または "y" 以外を入力すると置き換えない
     2: which reads & outterpretes user keyboard output
よろしいですか (Y/N)? y ……"y" またはリターンを入力すると実際に置き換えられる
     3: &  executes 'built-in' commands  like DIR or
よろしいですか (Y/N)? y
     3: &  executes 'built-in' comm&s  like DIR or
よろしいですか (Y/N)? n
*1,#l
     1:  COMMAND.COM,  the command interpreter
     2:*which reads & outterpretes user keyboard output
     3: &  executes 'built-in' commands  like DIR or
     4: initiates external PROGRAMs.
*■          "y" を答えたもののみが置き換えられている
```

(参考：Search)

Search

機能　指定された範囲内の行で，指定された文字列を検索します．
書式　[〈行番号 1〉][，〈行番号 2〉][？] S [〈文字列〉]

解　説

Search コマンドは，〈行番号 1〉と〈行番号 2〉で指定された範囲の行で〈文字列〉を検索し，〈文字列〉を含む最初の行を表示して，現在行を表示された行に移します．

①〈行番号 1〉，〈行番号 2〉S〈文字列〉

〈行番号 1〉と〈行番号 2〉の範囲で〈文字列〉を検索して，〈文字列〉を含む最初の行に現在行を移します．

```
*1l↵
     1:*                SYSTEM CALLS
     2:
     3: 1.1  INTRODUCTION
     4: MS-DOS provides two types of system calls:
     5: interrupts  and  function  requests.  This
     6: chapter  describes  the environments  from
     7: which these routines can be called ,how to
     8: call them, and the processing by each.
     9:
    10: 1.2  PROGRAMMING CONSIDERATIONS
    11: The system calls  mean  you  don't have to
    12: invent  your  own  ways  to  perform these
    13: primitive functions, and make it easier to
    14: write machine-independent programs.
*1,10show↵……第1～10行の間で "how" を検索する
     7: which these routines can be called ,how to……最初に一致した行が表示される
*……ここで"↵"とするとこの行を編集できる．↵のみだと次の行の編集モードにはいることができる
```

②［，〈行番号 2〉］R〈文字列〉

〈行番号 1〉を省略すると，現在行のすぐ後ろの行番号が指定されたとみなされます．

```
*,12sto ……現在行の次から第12行までの間で "to" を検索する(現在行は第8行とする)
    11: The system calls  mean  you  don't have to
*■
```

③［〈行番号 1〉］S〈文字列〉

〈行番号 2〉を省略すると，ファイルの最後の次の行〝#〟が指定されたとみなされます．

```
*6sand↵……第6行以下で "and" を検索する
     8: call them, and the processing by each.
*■
```

④ [〈行番号 1〉] [, 〈行番号 2〉] ? S 〈文字列〉

S の前に〝?〟を付けると，指定された〈文字列〉を含む行を表示し，〝よろしいですか〈Y／N〉?〟とユーザーに聞いてきます．そこで，〝Y〟かリターンキーを押すとその行が現在行となって検索は終了し，逆にそれ以外のキーを押すと次にマッチする行が見つかるか，指定された範囲の検索が終了するまで検索が続けられます．

```
*1,#?sthe.……第1行から終わりまでの間で "the" を?オプションを付けて検索する
     6: chapter  describes  the environments  from
よろしいですか <Y/N>? n……"n" または "y" 以外を押すと次を検索する
     7: which these routines can be called ,how to
よろしいですか <Y/N>? n
     8: call them, and the processing by each.
よろしいですか <Y/N>? y……"y" またはリターンを押すとそこで検索を終わる
*■
```

なお，MS-DOS ver.2.0 の EDLIN では，同じ文字列を連続して検索する際，2 回目以降は〝S〟のみを入力すれば目的の結果が得られます．

Transfer

機能　指定されたディスク上のファイルを〈行番号〉で指定された行の直前に読み込み，連結します．

書式　[〈行番号〉] T 〈ファイル名〉

解　説

Transfer コマンドは，〈ファイル名〉で指定されたディスク上のファイルを，〈行番号〉で指定された行の直前に読み込み連結します．〈行番号〉を省略すると，現在行が省略されたとみなされます．指定したファイルを読み込むと，行番号は自動的に付け換わります．

```
A>edlin mes.c……EDLINで "MES . C" を作る
新しいファイルです
*i……インサートモードにはいる
     1:*main()
     2:*{
     3:*        cls();
     4:*        locate(15,8);
     5:*        printf('MS-DOS ver 2.11');
     6:*}
     7:*^Z……CTRL-Z. を入力してインサートモードを抜ける
*7tlocate.c……第7行の直前(ファイルの最後)にあらかじめ作っておいた "LOCATE . C" を読み込んで連結する
*1,#l
     1: main()
     2: {
     3:         cls();
     4:         locate(15,8);
     5:         printf('MS-DOS ver 2.11');
     6: }
```

```
     7:*locate(x,y)
     8: int x,y;
     9:
    10: {
    11:         printf('¥x1b[%d;%dH',y,x);
    12: }
    13:
    14: cls()
    15: {
    16:         printf('¥x1b[2J');
    17: }
*■
```

連結されている

Write

機能　メモリ上のファイルから〈n〉行だけ出力ファイルに書き出します．

書式　[〈n〉] W

解　説

Write コマンドは，Append コマンドと併せて，メモリにはいりきらないファイルを編集する際に用います．編集しようとするファイルが，メモリにはいりきらない場合には，それを少しずつメモリにはいる分だけ読み込んできて，それについて編集を行い，編集の終わった部分をディスクに書き込んだ後に，はいりきらなかった分を読み込んできて編集を行います．

メモリにはいりきらないファイルを指定して EDLIN を起動すると，メモリの 4 分の 3 がいっぱいになるまで読み込まれます．ディスクに残っている部分を編集するためには，この Write コマンドによって編集の済んだ分を先頭から n 行ディスクへ書き込んでメモリに空き領域を作り，続いて Append コマンドによってファイルの残りをディスクから読み込みます．

Write コマンドを実行すると，ディスクへ書き込まれた分はメモリからなくなり，メモリ内のファイルについて 1 から順に行番号が付けられます．

〈n〉を省略すると，メモリの 4 分の 3 がディスクに書き込まれます．なお，使用可能なメモリが 4 分の 3 以上ある場合，Write コマンドを実行しても何も行われません．

```
A>edlin srchdir.asm□……"SRCHDIR．ASM" をEDLINで編集する
*1□……メモリがいっぱいになったので "ファイルを終りまで読込みました" というメッセージは出力されない
     1:*         page    255,80
     2:
     3: ;--------------------------------;
     4: ;        Directory search        ;
     5: ;--------------------------------;
     6: ;
     7: MAXLVL   equ     4
     8: DTASIZE  equ     43
     9: CHRLEN   equ     2
    10: ;
    11: ;   definition of stack frame
                    ment word public 'stack'
                                ;third argument
```

```
    21: spframe ends
    22: stack   ends
    23: ;
*10,12d ……第10～第12行を削除する
*w ……編集の終わった部分をファイルの先頭から出力ファイルに書き込んでメモリに空き領域を作る．その行数はEDLINが自動的に決定する
*a ……メモリにはいりきらなかった分をディスクからメモリに読み込む
*l ……まだ全部はいりきらない．再びメモリ上のファイルを編集する
    1:*         cmp     start,0
    2:          jne     tonext
    3:
    4:          push    ds
    5:          mov     ax,es
    6:          mov     ds,ax
                mov     ah,4eh
                mov     cx,1eh
   17:
   18: name_lp:mov      al,[bx
   19:         cmp      al,0
   20:         je       nxtfile
   21:         cmp      al,'.'
   22:         jne      tonext
   23:         inc      bx
*8
    8:*         mov     si,arg4[bp]
    8:*         mov     si,arg4[bx]
*500w ……先頭から500行出力ファイルに書き込む
*a ……メモリにはいりきらなかった分をメモリに読み込む
ファイルを終りまで読込みました……読込み中ファイルが終わりに達したというメッセージが出力されている
*l
    1:*         jmp     name_lp
    2:
    3: tonext:  or      start,80h
    4:
    5:          test    byte ptr [15h][di],10h
    6:          je      not_dir
                        dx,2
   17:
   18: nxtfile:mov      ah,
   19:         int     21h
   20:         jc      no_more
   21:         test    start,80h
   22:         je      name_ck
   23:         jmp     return
*e ……EDLINを終了する

A>■
```

（参考：Append コマンド）

3章 DEBUG

DEBUGの起動

DEBUGの起動方法は次の2通りがあります．

(1) DEBUGのみをロードする

A > DEBUG ↵

上のように入力すると，DEBUGがメモリにロードされ，プロンプト〝－〟が表示されてコマンドの入力待ちになります．目的のファイルをロードする際には，Nameコマンドおよび Loadコマンドを用います．

(2) DEBUGと同時に目的のファイルをロードし，目的のファイル実行時のパラメータをセットする

A > DEBUG 〈ファイル名〉［〈パラメータ〉］ ↵

〈ファイル名〉：デバッグしようとするプログラムのファイル名
〈パラメータ〉：そのプログラム実行時に必要なパラメータ類

上のように入力すると，DEBUGがメモリにロードされた後，DEBUGは使用可能なメモリの最下位セグメントのオフセット100H番地以降に〈ファイル名〉で指定されたファイルをロードし，BX：CXレジスタにロードしたファイルのバイト数をセットします．またパラメータ類もセットしますから，DOSのコマンドレベルからコマンドを入力したかのごとくDEBUG上で，ロードしたファイルを実行することができます．

注意：MS-DOS ver. 2.0用のDEBUGは，PSPをDEBUGの直後，空き領域にセットします．COMファイル，EXEファイルのデバッグを行う際には，PSPのオフセット5BH以前を変更することはできません．

DEBUGのコマンド

DEBUGのコマンドは，アルファベット1文字のコマンド名と，パラメータの組み合せから構成されます．MS-DOSのコントロール・キャラクタとテンプレート機能は，そのままDEBUG内でも使用できます．入力は大文字でも小文字でもかまいません．

パラメータ

Quit コマンド以外のコマンドは，パラメータを伴います．パラメータは，スペースまたはカンマによって区切ります．

〈ドライブ〉　読出し／書込みを行うディスク・ドライブを 1 桁の 16 進数で指定します．ドライブ名(A：～)と番号の対応は，システムによって異なります．

〈バイト〉　メモリやレジスタの内容を表す 2 桁の 16 進数です．

〈レコード〉　ディスク上の論理レコード番号で，Load コマンド，Write コマンドで読み書きを行うディスクセクタで，1～3 桁の 16 進数で指定します．論理レコード番号とは，トラック 0 のセクタ 1 を 0 とし，順次最終トラックの最終セクタまで付けられた番号です．

［例］　たとえば両面，9 セクタ／トラックのディスクの場合，論理レコードは Table 3.1 のようになる

Table 3.1　論理レコード番号(両面 9 セクタ／トラックの場合)

トラック	0		1		2		…
セクタ ＼ サーフェス	0	1	0	1	0	1	…
1	0	9	18	27	36	45	⋮
2	1	10	19	28	37	46	
3	2	11	20	29	38	47	
4	3	12	21	30	39	48	
5	4	13	22	31	40	49	
6	5	14	23	32	41	50	
7	6	15	24	33	42	51	
8	7	16	25	34	43	52	
9	8	17	26	35	44	53	

〈値〉　ポート番号やコマンドの繰返し回数を指定するための 4 桁までの 16 進数です．

〈アドレス〉　アルファベットによるセグメント・レジスタ指定，もしくは 4 桁までの 16 進数のセグメント・アドレスと，4 桁までの 16 進数のオフセットの 2 つの部分から成り，セグメント指定とオフセット指定を〝：〟で区切ります．セグメント指定を省略すると，デフォルトのセグメント・レジスタの内容が使われます．デフォルトのセグメント・レジスタはコマンドによって異なり，Assemble, Go, Load, Trace, Unassemble, Write コマンドでは CS レジスタ，それ以外はすべて DS レジスタです．

［例］　163A：0100　セグメント・アドレス 163AH オフセット 100H

CS：0100　CS レジスタの内容で示されるセグメント・アドレスのオフセット 100H

〈レンジ〉　メモリ・アドレスの範囲を指定するもので，2通りの指定方法があります．

①〈アドレス1〉〈アドレス2〉

〈アドレス1〉で開始アドレスを，〈アドレス2〉で終了アドレスを指定します．なお，〈アドレス2〉にはセグメント・アドレスの指定はできず，〈アドレス1〉で指定したセグメント・アドレスが用いられます．

②〈アドレス〉　L〈値〉

〈アドレス〉で開始アドレスを，〈値〉でバイト数を指定します．〈値〉の最大値は10000Hまでで，10000Hを指定する場合は0を用います．

[例]　152D：200　FFF　アドレス152D：200Hから152D：FFFHまで

CS：100　1FF　アドレスCS：100HからCS：1FFHまで

CS：100　L100　アドレスCS：100Hから長さ100Hバイト

（CS：100　1FFと同様）

〈リスト〉　任意個の〈バイト〉，または〈ストリング〉で指定します．

〈リスト〉はコマンドラインの最後に付きます．

〈ストリング〉　引用符(' 'または" ")によって囲まれた任意長の文字列です．Table 3.2に示すように，引用符はシングル(' ')でもダブル(" ")でもかまいませんが，先頭に用いた方のみが引用符として認識され，もう一方は普通のアルファベットや記号と同じように扱われます．引用符自体を〈ストリング〉の中に含めたいときには，引用符を2つ並べます．

Table 3.2　ストリングの指定

キーボードからの入力	コマンドが実際に使用する文字列
'MS-DOS'	MS-DOS
"MS-DOS"	MS-DOS
'abc"def"ghi'	abc"def"ghi
"abc""def""ghi"	abc"def"ghi
"abc'def'ghi"	abc'def'ghi
'abc''def''ghi'	abc'def'ghi
'abc'def'ghi'	誤
"abc"def"ghi"	誤

Assemble

機能　8086(8088)／8087 の命令を 1 行ずつアセンブルして，指定した〈アドレス〉以降のメモリに収めます．

書式　A［〈アドレス〉］

解　説

①A〈アドレス〉

指定した〈アドレス〉から 8086(8088)／8087 の命令を入力し，1 ラインごとにアセンブルしてメモリに書き込みます．Assemble コマンドを終了するには，CTRL-C または何も入力せずにリターンキーを押します．

```
-a1000↵……１０００H番地からアセンブルする．セグメント値はDEBUGが自動的に決めてくれる
1626:1000 mov ah,02↵……1行命令を入力してリターンを押すと，それに対応したコードがメモリに直接 書き込まれる
1626:1002 mov dl,41↵
1626:1004 int 21↵
1626:1006 int 20↵
1626:1008↵……リターンのみを入力してDEBUGのコマンド待ちに戻る
-■
```

②A

〈アドレス〉を省略した場合は，IP の値(Register コマンドによって表示される IP の値)を〈アドレス〉とみなします．

```
A>debug↵
-r↵……レジスタの内容を表示する
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1626  ES=1626  SS=1626  CS=1626  IP=0100   NV UP DI PL NZ NA PO NC
1626:0100 45              INC     BP
-a↵……aのみを入力するとDEBUG起動時にはCS：１００Hから(CS：IPの値から)アセンブルされる
1626:0100 mov ah,9↵
1626:0102 mov dx,109↵
1626:0105 int 21↵
1626:0107 int 20↵
1626:0109
-a↵……次にまたa↵を入力すると直前に実行したAssembleコマンドの終了アドレス(CS：１０８H)の次からアセンブルされる
1626:0109 db 'MS-DOS Hand book$'↵
1626:011A↵
-■
```

Assemble コマンドの文法上の注意

Assemble コマンドを使用するには，次のような注意事項があります．

□ Assemble コマンドで入力できる数値は，すべて 1〜4 桁の 16 進数です．

□ プリフィックス命令(REP，LOCK あるいはセグメント・オーバーライドなど)はオペコードの

前に置かなければなりません．また，別ラインに分けて入力することもできます．

□セグメント・オーバーライドは〝CS：〟，〝DS：〟，〝ES：〟，〝SS：〟で指定します．

□ストリング操作命令(MOVS, LODS など)を入力する場合は，ストリングサイズ(バイトまたはワード)を明確に指定しなければなりません．たとえば，ワード・ストリングの移動にはMOVSW，バイト・ストリングの移動にはMOVSB を用います．

□ FAR リターン命令は RETF です．

□ JMP，CALL 命令では，指定されたオペランドによって自動的に SHORT，NEAR，FAR が決定され，アドレスに変換されます．ただし，NEAR または FAR のプリフィックスを置いた場合はそれに従います．なお，NEAR プリフィックスは〝NE〟と略記することができます．

□オペランドがワード単位，バイト単位のどちらであるかが直接判断できない場合は，〝WORD PTR〟，〝BYTE PTR〟のプリフィックスによってデータタイプを明確に指定します．なお，WORD PTR は〝WO〟，BYTE PTR は〝BY〟と略記することができます．

□メモリを参照する場合は，角カッコ〝[ ]〟でロケーションを指定します．たとえば，メモリ・ロケーション 3256H の内容を AX レジスタにロードする命令は〝MOV AX，[3256]〟と書きます．

□擬似命令〝DB〟および〝DW〟が使用できます．DB はバイト値を，DW はワード値を直接メモリへ書き込みます．

□レジスタ間接指定の表記が使用できます(〝ADD BX，34 [BP+2] [SI-1]〟，〝POP [BP+DI]〟，〝PUSH [SI]〟など)．

□すべての同義オペコードの表記が使用できます(〝LOOPZ〟と〝LOOPE〟あるいは〝JA〟と〝JNBE〟など)．

□ 8087 のオペコードは，明確に指定しなければなりません．たとえば〝WAIT〟は 8086 のオペコードであり，8087 は〝FWAIT〟と書きます．

(参考：Unassemble)

Compare

機能 〈レンジ〉で指定した範囲のメモリの内容と，〈アドレス〉ではじまる同じ大きさの範囲のメモリの内容とを比較します．

書式 C〈レンジ〉〈アドレス〉

解 説

2つのメモリ領域の内容が一致している場合は何も表示されません．もし違いがあるときは，次のように異なる部分のアドレスとその内容を表示します．

```
-c2000:0 7f 100↵ ……2000：0Hから2000：7FHまでの内容とCS：100Hから同じ大きさの範囲のメモリの内容を比較する
2000:0003  61  41  1626:0103
2000:000A  64  44  1626:010A
2000:0013  70  50  1626:0113
2000:001A  65  45  1626:011A
2000:0023  6E  4E  1626:0123   } 内容の異なるアドレスとその内容が表示される
2000:002F  6D  4D  1626:012F
2000:0036  65  45  1626:0136
2000:003C  64  44  1626:013C
-d2000:0↵ ……2000：0Hから80Hバイトダンプする
2000:0000  0D 0A 24 61 63 63 65 73-73 20 64 65 6E 69 65 64   ..$access denied
2000:0010  0D 0A 24 70 61 72 69 74-79 20 65 72 72 6F 72 20   ..$parity error
2000:0020  6F 72 20 6E 6F 6E 65 78-69 73 74 61 6E 74 20 6D   or nonexistant m
2000:0030  65 6D 6F 72 79 20 65 72-72 6F 72 20 64 65 74 65   emory error dete
2000:0040  63 74 65 64 0D 0A 24 00-00 00 00 00 00 00 00 00   cted..$.........
2000:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2000:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2000:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-d100↵ ……CS：100Hから80Hバイトダンプする
1626:0100  0D 0A 24 41 63 63 65 73-73 20 44 65 6E 69 65 64   ..$Access Denied
1626:0110  0D 0A 24 50 61 72 69 74-79 20 45 72 72 6F 72 20   ..$Parity Error
1626:0120  6F 72 20 4E 6F 6E 65 78-69 73 74 61 6E 74 20 4D   or Nonexistant M
1626:0130  65 6D 6F 72 79 20 45 72-72 6F 72 20 44 65 74 65   emory Error Dete
1626:0140  63 74 65 64 0D 0A 24 00-00 00 00 00 00 00 00 00   cted..$.........
1626:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1626:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1626:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-■
```

Dump

機能　メモリの内容の16進ダンプとASCIIダンプを表示します.

書式　D［〈レンジ〉］

解　説

①D〈レンジ〉

〈レンジ〉で指定される範囲のメモリの内容をダンプします.

```
A>debug command.com↵ ……DEBUGを起動して"COMMAND.COM"をロードする
-d3dcf 3edf↵ ……3DCFHから3EDFHまでをダンプする
163F:3DC0                                                  45                E
163F:3DD0  52 52 4F 52 4C 45 56 45-4C E4 0D 05 45 58 49 53   RRORLEVELd..EXIS
163F:3DE0  54 A6 0D 00 04 44 49 52-01 A8 0F 07 52 45 4E 41   T&...DIR.(..RENA
163F:3DF0  4D 45 01 A2 11 04 52 45-4E 01 A2 11 06 45 52 41   ME.'..REN.'..ERA
163F:3E00  53 45 01 7A 11 04 44 45-4C 01 7A 11 05 54 59 50   SE.z..DEL.z..TYP
163F:3E10  45 01 E1 11 04 52 45 4D-00 04 01 05 43 4F 50 59   E.a..REM....COPY
163F:3E20  01 66 1D 06 50 41 55 53-45 00 6D 11 05 44 41 54   .f..PAUSE.m..DAT
163F:3E30  45 00 34 14 05 54 49 4D-45 00 0E 15 04 56 45 52   E.4..TIME....VER
163F:3E40  00 9F 12 04 56 4F 4C 01-50 12 03 43 44 01 05 13   ....VOL.P..CD...
163F:3E50  06 43 48 44 49 52 01 05-13 03 4D 44 01 38 13 06   .CHDIR....MD.8..
163F:3E60  4D 4B 44 49 52 01 38 13-03 52 44 01 6D 13 06 52   MKDIR.8..RD.m..R
163F:3E70  4D 44 49 52 01 6D 13 06-42 52 45 41 4B 00 FD 1B   MDIR.m..BREAK.}.
163F:3E80  07 56 45 52 49 46 59 00-2F 1C 04 53 45 54 00 C4   .VERIFY./..SET.D
163F:3E90  08 07 50 52 4F 4D 50 54-00 AE 08 05 50 41 54 48   ..PROMPT....PATH
163F:3EA0  00 27 1A 05 45 58 49 54-00 BF 0A 05 43 54 54 59   .'..EXIT.?..CTTY
163F:3EB0  01 DA 0A 05 45 43 48 4F-00 B3 1B 05 47 4F 54 4F   .Z..ECHO.3..GOTO
163F:3EC0  00 53 0E 06 53 48 49 46-54 00 12 0E 03 49 46 00   .S..SHIFT....IF.
163F:3ED0  0C 0D 04 46 4F 52 00 23-0C 04 43 4C 53 00 F3 12   ...FOR.#..CLS.s.
-■
   └─オフセット
   └──セグメント
```

②D〈開始アドレス〉

〈開始アドレス〉のみを指定した場合は，そこから128バイトを表示します．

```
A>debug link.exe↵……DEBUGを起動して"LINK．EXE"をロードする
-d100↵……100Hからダンプする．終了アドレスを省略するとそのアドレスから80Hバイトダンプされる
163F:0100  4C 69 63 65 6E 73 65 64-20 4D 61 74 65 72 69 61   Licensed Materia
163F:0110  6C 20 2D 20 50 72 6F 67-72 61 6D 20 50 72 6F 70   l - Program Prop
163F:0120  65 72 74 79 20 6F 66 20-4D 69 63 72 6F 73 6F 66   erty of Microsof
163F:0130  74 20 49 6E 63 2E 33 C0-F7 D0 CA 02 00 00 55 8B   t Inc.3@wPJ...U.
163F:0140  EC 83 EC 0E 81 3E D0 09-FF 00 73 08 81 3E CE 09   l.l..>P...s..>N.
163F:0150  FF 00 72 0D B8 00 00 50-B8 A0 00 50 9A B1 01 AA   ..r.8..P8 .P.1.*
163F:0160  17 9A 46 01 76 17 88 46-F6 80 7E F6 00 75 10 9A   ..F.v..Fv.~v.u..
163F:0170  17 02 76 17 89 46 F8 9A-46 01 76 17 88 46 F7 9A   ..v..Fx.F.v..Fw.
-■
```

③D

パラメータをすべて省略した場合は，直前に実行したDumpコマンドの終了アドレスの次から128バイト(DEBUG起動時はDS：100Hから128バイト)を表示します．

```
A>debug sys.com↵……DEBUGを起動して，"SYS．COM"をロードする
-d↵……DEBUG起動時にDのみを入力するとDS：100Hから80Hバイトダンプされる
163F:0100  EB 09 56 65 72 73 20 31-2E 38 31 50 B4 30 CD 21   k.Vers 1.81P40M!
163F:0110  86 E0 3D 36 01 72 05 3D-0B 02 76 09 BA 1C 04 B4   .`=6.r.=..v.:..4
163F:0120  09 CD 21 CD 20 58 EB 2F-BA 00 04 8B 0E 1A 04 E9   .M!M Xk/:......i
163F:0130  45 01 BA E6 03 8B 0E FE-03 E9 3B 01 A0 83 04 04   E.:f...~.i;. ...
163F:0140  40 A2 54 04 BA 39 04 8B-0E 81 04 BB 02 00 B4 40   @"T.:9.....;..4@
163F:0150  CD 21 E8 17 02 32 C0 80-3E 5D 00 20 75 CA 3C FF   M!h..2@.>]. uJ<.
163F:0160  74 D0 80 3E 5C 00 00 74-C9 B4 19 CD 21 FE C0 A2   tP.>¥..tI4.M!~@"
163F:0170  83 04 38 06 5C 00 74 BA-04 40 A2 84 04 A2 8E 04   ..8.¥.t:.@"..".
-d↵……次にDのみを入力すると直前に実行したDumpコマンドの終了アドレス(DS：17FH)の次から80Hバイトダンプされる
163F:0180  FC BA 84 04 BF 9B 04 E8-4D 01 72 B0 BA 8E 04 BF   ¦:..?..hM.r0:..?
163F:0190  AB 04 E8 42 01 72 A5 8B-CC 81 E9 C8 06 89 0E C2   +.hB.r%.L.iH...B
163F:01A0  04 E8 D7 00 A0 5C 00 04-40 A2 84 04 A2 8E 04 A2   .hW. ¥..@"..".."
163F:01B0  BB 04 B4 4E BA BB 04 B9-06 00 CD 21 72 29 8A 16   ;.4N:;.9..M!r)..
163F:01C0  5C 00 B4 36 CD 21 F7 E1-91 BA 84 04 A1 9F 04 8B   ¥.46M!wa.:..!...
163F:01D0  1E A3 04 E8 5F 01 75 55-BA 8E 04 A1 AF 04 8B 1E   .#.h_.uU:..!/...
163F:01E0  B3 04 E8 50 01 77 46 BA-84 04 B9 00 00 B8 01 43   3.hP.wF:..9..8.C
163F:01F0  CD 21 BA 8E 04 B9 00 00-B8 01 43 CD 21 BA 84 04   M!:..9..8.CM!:..
-■
```

なお，コントロールコードなど，ASCIIキャラクタで表示できないものはピリオド〝.〟が表示されます．

Enter

機能　指定された〈アドレス〉のメモリの内容を変更します．
書式　E〈アドレス〉[〈リスト〉]

解　説

①E〈アドレス〉〈リスト〉

〈アドレス〉と〈リスト〉の両方を指定すると，〈アドレス〉からはじまるメモリの内容が〈リスト〉の値に書き換えられます．

```
A>debug
-e100 4d 53 2d 44 4f 53          ┌アドレスCS：100Hから順に4DH，53H，2DH，44H，4FH，53Hを書き込む．
                                 └この場合E100' MS-DOS' でもよい
-d100 10f ……100H～10FH番地までダンプしてみる
1626:0100  4D 53 2D 44 4F 53 49 4D-45 00 0E 15 04 56 45 52   MS-DOSIME....VER
-■                                  Enterコマンドによって書き込んだ値
```

②E〈アドレス〉

〈アドレス〉のみを指定すると，そのアドレスを表示して入力待ちになりますから，新しい値を16進数で入力します．次のアドレスに進むにはスペースキーを，前のアドレスに戻るには〝－〞を押します．またEnterコマンドを終了するにはリターンキーを押します．

```
-e100 ……〈リスト〉を省略してE〈アドレス〉(E100)のみを入力すると アドレスを表示して入力待ちになる
1626:0100  45.21   00.00   34.35   14.-……〝－〞を入力するとアドレスが1つ戻って改行される
1626:0102  35.36   14.     05.37
-d100 10f                          リターンを押すとDEBUGのコマンドレベルへ戻る
1626:0100  21 00 36 14 37 54 49 4D-45 00 0E 15 04 56 45 52   !.6.7TIME....VER
-■                  変更の必要がないときはスペースキーのみを押す
                    値を16進数で入力しスペースキーを押すと次のアドレスへ進む
```

Fill

機能　〈レンジ〉で指定された範囲のメモリを，〈リスト〉で指定された値で満たします．

書式　F〈レンジ〉〈リスト〉

解　説

〈リスト〉内の値の個数よりも，〈レンジ〉のバイト数の方が多い場合には，〈レンジ〉のすべてのメモリが書き換え終わるまで，〈リスト〉内の値を繰り返し使います．

```
A>debug
-f100 11f 0 …… 100H～11FH番地を0で埋める
-d100 11f
1626:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1626:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-f120 150 41 42 43 44 …… 120H～150H番地を41H，42H，43H，44Hの繰返しで埋める
-d120 15f
1626:0120  41 42 43 44 41 42 43 44-41 42 43 44 41 42 43 44   ABCDABCDABCDABCD
1626:0130  41 42 43 44 41 42 43 44-41 42 43 44 41 42 43 44   ABCDABCDABCDABCD
1626:0140  41 42 43 44 41 42 43 44-41 42 43 44 41 42 43 44   ABCDABCDABCDABCD
1626:0150  41 56 45 52 49 46 59 00-2F 1C 04 53 45 54 00 C4   AVERIFY./..SET.D
-■         Fillコマンドによって41H，42H，43H，44Hの繰返しで埋められた
```

〈レンジ〉のバイト数よりも，〈リスト〉内に並んだ値の個数の方が多い場合には，〈リスト〉内の余分な値は無視されます．

```
-d100 10f
1626:0100  45 00 34 14 05 54 49 4D-45 00 0E 15 04 56 45 52   E.4..TIME....VER
-f100 107 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f ……100H～107H番地を40H，41H，…4FHで埋める
-d100 10f
1626:0100  40 41 42 43 44 45 46 47-45 00 0E 15 04 56 45 52   @ABCDEFGE....VER
-■
          41H～47Hまでが書き込まれたが48H～4FHは〈レンジ〉からあふれているので無視される
```

〈レンジ〉内のメモリ中に，使用不可能なもの(不良またはメモリが実装されていない場合)があると，エラーはそれに続くすべてのアドレスに広がってしまいます．
(参考：Enter)

Go

機能　DEBUGの管理下で，メモリ上にロードされたプログラムを実行します．〈アドレス1〉はスタート・アドレスを，〈アドレス2〉……はブレーク・ポイントを設定するものです．

書式　G［＝〈アドレス1〉］［〈アドレス2〉……］

解　説

①G＝〈アドレス1〉

〝＝〈アドレス1〉〟を指定すると，そのアドレスから実行がはじまります．〝＝〈アドレス1〉〟のうち，セグメント指定を省略するとIPのみが設定され，CSレジスタはそのときの値(Registerコマンドによって表示されるCSレジスタの値)が用いられます．

```
A>debug
-a1000 ……1000H番地からプログラムをAssembleコマンドでメモリに書き込む
1624:1000 mov ah,09
1624:1002 mov dx,1009
1624:1005 int 21
1624:1007 int 20
1624:1009 db 'MS-DOS DEBUG.COM $'
1624:101B
-g=1000 ……1000H番地からプログラムを実行する
MS-DOS DEBUG.COM
Program terminated normally   うまく実行されている
-■
          プログラムが正常に終了するとDEBUGが出力する
```

②G

パラメータをすべて省略すると，CSレジスタとIPの値の示すアドレスからプログラムが実行されます．通常(DEBUG起動時やLoadコマンドでファイルのロード時)は，プログラムの先頭にCSレジスタおよびIPが設定されていますので，プログラムは先頭から実行されます．

```
A>dir fact↵

 ドライブ A: のボリュームラベルは PC-9801
 ディレクトリは A:¥

FACT     EXE     13496  84-03-05  19:14
       1 個のファイルがあります
   777216 バイトが使用可能です

A>debug fact.exe 10↵ ……DEBUGを起動して"FACT.EXE"をその管理下にロードする．この場合"FACT.EXE"のパラメータ10も一緒に書いておくとDEBUGからGoコマンドで実行したときそのパラメータが使用され，DOSのコマンドレベルからFACT 10と入力したのと同じ結果が得られる
-g↵ ……Goコマンドで実行する．アドレスを省略するとプログラムの先頭から実行される
10!=3628800
Program terminated normally ……正常にプログラムが終了した
-■
```

③G＝〈アドレス1〉〈アドレス2〉‥‥

"＝〈アドレス1〉〈アドレス2〉‥‥"を指定すると，〈アドレス1〉からプログラムの実行がはじまり，"〈アドレス2〉‥‥"のアドレスで実行が停止され，そのときのレジスタ，フラグの内容およびブレーク・ポイント・アドレスのニーモニック表記の命令が表示されます(Registerコマンドと同じものが表示される)．ブレーク・ポイントは最高10か所まで設定でき，設定できる場所は8086(8088)のオペコードの先頭アドレスのみです．

```
A>dir func-0f↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥asmwork

FUNC-0F  ASM      1517  84-03-06  20:09
FUNC-0F  COM       247  84-03-06  20:10 ……ファイルのコピーをするサンプル・プログラム
FUNC-0F  LST      4616  84-03-06  20:10
       3 個のファイルがあります
   722944 バイトが使用可能です

A>debug func-0f.com func-0f.lst test↵ ……"DEBUG"を起動して"FUNC-0F.COM"をロードするDEBUGの管理下で"FUNC-0F.LST"を"TEST"にコピーする
-u↵                  └コピーのソース └ディスティネーション
163F:0100 B91000        MOV     CX,0010
163F:0103 BE6C00        MOV     SI,006C
163F:0106 BFD201        MOV     DI,01D2
163F:0109 1E            PUSH    DS
163F:010A 07            POP     ES
163F:010B FC            CLD
163F:010C F3            REPZ
163F:010D A4            MOVSB
163F:010E B40F          MOV     AH,0F
163F:0110 BA5C00        MOV     DX,005C
163F:0113 CD21          INT     21
163F:0115 3CFF          CMP     AL,FF ……ブレーク・ポイントを置くアドレス〈ファンクション0FHのシステムコール(ファイルのオープン)からの戻り値を確認するため〉
163F:0117 742F          JZ      0148
163F:0119 B416          MOV     AH,16
163F:011B BAD201        MOV     DX,01D2
163F:011E CD21          INT     21
-g=100 115↵ ……115H番地にブレーク・ポイントを置き100H番地からプログラムを実行する(プログラムは100H～114Hまで実行されることになる)
      └AL=00Hになっている．ファイル"FUNC-0F.LST"が存在している
AX=0F00  BX=0000  CX=0000  DX=005C  SP=FFFE  BP=0000  SI=007C  DI=01E2
DS=163F  ES=163F  SS=163F  CS=163F  IP=0115   NV UP DI PL NZ NA PO NC
163F:0115 3CFF          CMP     AL,FF
-g↵ ……プログラムの実行がブレーク・ポイント・アドレスに達するとレジスタを表示してDEBUGのコマンド待ちになる

Copy Complete ……プログラム"FUNC-0F.COM"からのメッセージ

Program terminated normally ……プログラムが正常に終了したというDEBUGからのメッセージ
-q↵ ……DEBUGから抜ける
```

```
A>dir test↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥asmwork

TEST             4736  84-03-23   5:47 ……ファイル"TEST"ができている("FUNC-0F.COM"についての詳細はシステムコールのサンプルを参照)
       1 個のファイルがあります
  717824 バイトが使用可能です

A>■
```

④ G 〈アドレス 2〉 ····

ブレーク・ポイントを〝〈アドレス 2〉 ····〟に設定し，CS レジスタおよび IP で示されるアドレスからプログラムを実行します．

```
 >debug chkdsk.com↵
-u↵
1642:0100 EB08          JMP     010A
1642:0102 56            PUSH    SI
1642:0103 65            DB      65
1642:0104 7220          JB      0126
1642:0106 322E3330      XOR     CH,[3033]
1642:010A 50            PUSH    AX
1642:010B B430          MOV     AH,30
1642:010D CD21          INT     21
1642:010F 86E0          XCHG    AL,AH
1642:0111 3D3601        CMP     AX,0136
1642:0114 7205          JB      011B ……ブレーク・ポイントを置くアドレス．ファンクション30Hのシステムコール
1642:0116 3D0B02        CMP     AX,020B(DOSのバージョン番号を得る)からの戻り値をチェックする
1642:0119 7606          JBE     0121
1642:011B BA6014        MOV     DX,1460
1642:011E EB31          JMP     0151
-g114↵……ブレークポイントを114H番地にしてプログラムを実行する．実行アドレスを省略するとプログラムの先頭(CS：100H番地)から実行がはじまる
      ┌DOSのバージョンは2．11(注：10FH番地でXCHG AL，AHを実行している)
AX=020B  BX=FF00  CX=0000  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=1642  ES=1642  SS=1642  CS=1642  IP=0114   NV UP DI PL NZ NA PO NC
1642:0114 7205          JB      011B                                  └CYが立っていないから次のJB命令で分岐は起こらない
-g119↵……ブレークポイントを119H番地にしてプログラムを実行する
        この場合前の続き(114H番地から)が実行できる
AX=020B  BX=FF00  CX=0000  DX=0000  SP=FFFC  BP=0000  SI=0000  DI=0000
DS=1642  ES=1642  SS=1642  CS=1642  IP=0119   NV UP DI PL ZR NA PE NC
1642:0119 7606          JBE     0121
-g↵……プログラムをブレーク・ポイント・アドレスから実行する
ディスク PC9800_SYS  は 1983-10-31 19:27 に作成されました

  1250304 バイト : 全ディスク容量
    59392 バイト : 3 個のシステムファイル
     2048 バイト : 2 個のディレクトリ
   863232 バイト : 49 個のユーザーファイル
   325632 バイト : 使用可能ディスク容量

   262144 バイト : 全メモリ
   170976 バイト : 使用可能メモリ

                                  〝CHKDSK.COM〟からの出力

Program terminated normally
-■
```

実行上の注意

□ Go コマンドは，IRET 命令を使ってプログラムへジャンプするので，ユーザースタックは最低 6 バイト必要です．ユーザースタックが使用不能かあるいはそれより小さい場合，DOS が破壊される場合があります．

□指定されたブレーク・ポイント・アドレスには割込みコード CCH(INT　03H)が置かれ，実行時にブレーク・ポイントコードを持った命令の１つに出会うと，すべてのブレーク・ポイント・アドレスにもとの命令が再格納されます．なお，実行中どのブレーク・ポイントにも出会わずに停止した場合は，割込みコードはもとの命令に置き換えられません．

□一度実行が終了したプログラムを再実行する場合，そのプログラムによってメモリの内容が書き換えられているために再実行できないことがあります．この場合は，Name コマンドで FCB に実行したいプログラムのファイル名をセットした後，Load コマンドによってプログラムを再ロードする必要があります．

□ DEBUG は等号〝=〞によってスタート・アドレスとブレーク・ポイントを区別しているのでスタート・アドレスにはかならず等号を付けなければなりません．

(参考：Trace)

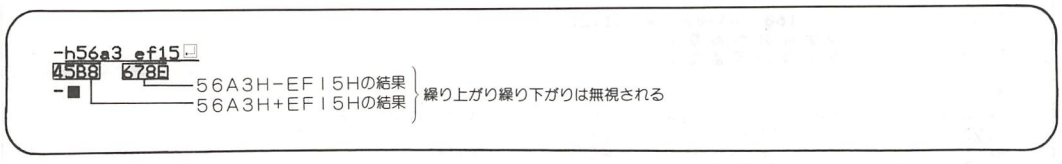
Hex

機能　２つの 16 進数の和と差を計算します．

書式　H 〈値 1〉〈値 2〉

解　説

〈値 1〉＋〈値 2〉および〈値 1〉－〈値 2〉を計算し，その順にスクリーンに表示します．DEBUG 実行中のアドレス計算などに利用します．



Input

機能　〈値〉で指定された I／O ポートから１バイト入力し，スクリーンに表示します．

書式　I 〈値〉

解　説

〈値〉には 16 ビットのポート・アドレスを指定します．

(参考：Output)

```
-i90↵……ポート・アドレス９０Ｈの内容を表示する
80
-i1036↵……ポート・アドレス１０３６Ｈの内容を表示する(ポート・アドレスは16ビットまで)
45
-■
```

Load

機能　ファイルまたはディスクの，レコードの内容をメモリにロードします．

書式　L［〈アドレス〉［〈ドライブ〉〈レコード〉〈値〉］］

解　説

①L 〈アドレス〉

〈アドレス〉のみを指定すると，FCB にセットされたファイル名のファイルが〈アドレス〉からロードされ，BX：CX に読み込んだバイト数がセットされます．なお，ファイルをロードするときは DEBUG 起動時，または Name コマンドによって事前にファイル名が指定されて(FCB にセットされて)いなければなりません．また EXE ファイルをロードする場合，ヘッダで指定されるアドレスへリロケートされるので，〈アドレス〉の指定は無効です．

```
A>dir test↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

TEST             160  84-02-29  21:21
       1 個のファイルがあります
  716800 バイトが使用可能です

A>debug↵
-ntest↵……ロードするファイルのファイル名 "TEST" をNameコマンドでFCBにセットする
-l2000↵……アドレス2000H番地からファイル "TEST" をロードする
-r↵                    ┌──ロードしたバイト数000000A0H/バイト
AX=0000  BX=0000  CX=00A0  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1629  ES=1629  SS=1629  CS=1629  IP=0100   NV UP DI PL NZ NA PO NC
1629:0100 009F1204        ADD     [BX+0412],BL                 DS:0412=00
-d2000↵……ロードしたファイルをダンプしてみる．
1629:2000  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
1629:2010  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
1629:2020  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
1629:2030  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
1629:2040  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
1629:2050  50 51 52 53 54 55 56 57-58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[¥]^_
1629:2060  60 61 62 63 64 65 66 67-68 69 6A 6B 6C 6D 6E 6F   `abcdefghijklmno
1629:2070  70 71 72 73 74 75 76 77-78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
-■    実際は209FHまで続いている
```

②L

パラメータをすべて省略すると，FCB にセットされたファイルが CS：100H からロードされます．その他の詳細は①と同じです．

```
A>debug command.com↵……DEBUGを起動して"COMMAND．COM" をロードする
-r↵
AX=0000  BX=0000  CX=4326  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1643  ES=1643  SS=1643  CS=1643  IP=0100   NV UP DI PL NZ NA PO NC
1643:0100 E9CD0B        JMP     0CD0
-g↵……プログラム"COMMAND．COM" を実行する

Command ﾊﾞｰｼﾞｮﾝ 2.11

A>dir /w↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM     KNJDIC   SYS     BIN             COM             AUTOEXEC BAT
CONFIG   SYS     FUNC             FACT     EXE    TEST            SUBDIR1
CAL              TXT              ASMWORK
      13 個のファイルがあります
   716800 バイトが使用可能です

A>exit↵…"COMMAND．COM" を抜け，DEBUGへ戻る

Program terminated normally
-l↵ ┌"COMMAND．COM" を再ロードする．FCBにはDEBUG起動時に "COMMAND．COM" がすでにセットされているので
    └Nameコマンドで再設定する必要はない．また，アドレス100H番地からプログラムはロードされる
-r↵
AX=0000  BX=0000  CX=4326  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=165D  ES=165D  SS=165D  CS=165D  IP=0100   NV UP DI PL NZ NA PO NC
165D:0100 E9CD0B        JMP     0CD0    └CSレジスタおよびIPはプログラムの先頭にセットされる
-■
```

③L〈アドレス〉〈ドライブ〉〈レコード〉〈値〉

ディスクのセクタを直接指定して読み込むこともできます．この場合は，すべてのパラメータをかならず指定し，その〈ドライブ〉の〈レコード〉からはじまる〈値〉だけのセクタを〈アドレス〉ではじまるメモリ領域に読み込みます．なお〈ドライブ〉は番号で指定しますが，ドライブA：，B：‥‥とその番号との対応はシステムによって異なります．〈値〉はロードされる内容の大きさが64Kバイト以下に設定しなければ，正しくロードされません．また，この〈値〉の最大値はディスクの種類(セクタサイズ)によって異なります．

```
          ロード・アドレス
          ドライブA
          レコード0から
          2セクタ
-l100 0 0 2↵……100H番地 以降にドライブAのレコード0から2セクタ(この場合8インチ2Dなので2Kバイト)ロードする
-d100↵
1629:0100  EB 1C 90 4E 45 43 20 32-2E 30 30 00 04 01 01 00   k..NEC 2.00.....
1629:0110  02 C0 00 D0 04 FE 02 00-08 00 02 00 00 00 33 C0   .@.P.~........3@
1629:0120  8E D8 8E C0 8E D0 BC 8E-02 FC BE 0B 00 2E AD 3D   .X.@.P<..¦>...-=
1629:0130  80 00 75 6C BE 5A 00 B8-00 0A CD 18 B4 0C CD 18   ..ul>Z.8..M.4.M.  ｝IPLの一部
1629:0140  0E 1F B8 00 A0 8E C0 BF-40 01 AC 0A C0 74 04 AA   ..8. .@?@.,.@t.*
1629:0150  47 EB F7 B0 06 E6 37 F4-EB FD 54 68 69 73 20 69   Gkw0.f7tk)This i
1629:0160  73 20 74 68 65 20 64 61-74 61 20 64 69 73 6B 00   s the data disk.
1629:0170  4E 6F 20 73 79 73 74 65-6D 20 66 69 6C 65 73 00   No system files.
-■
```

(参考：Name, Write)

Move

機能　〈レンジ〉で指定されたメモリ領域の内容を，指定された〈アドレス〉からはじまる領域へ移します．

書式　M〈レンジ〉〈アドレス〉

解　説

転送先のメモリ領域の一部が，もとのメモリ領域の一部と重なっている場合も，転送はつねにデータを失うことなしに正常に行われます．Move コマンドは，メモリの内容のコピーを行うだけで，移動元のメモリ領域に他のデータが書き込まれることはありません．

```
-d100 19f                                                    移動範囲
1629:0100  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
1629:0110  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
1629:0120  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
1629:0130  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
1629:0140  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
1629:0150  50 51 52 53 54 55 56 57-58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[¥]^_
1629:0160  60 61 62 63 64 65 66 67-68 69 6A 6B 6C 6D 6E 6F   `abcdefghijklmno
1629:0170  70 71 72 73 74 75 76 77-78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
1629:0180  80 81 82 83 84 85 86 87-88 89 8A 8B 8C 8D 8E 8F   ................
1629:0190  90 91 92 93 94 95 96 97-98 99 9A 9B 9C 9D 9E 9F   ................
-m140 15a 200      ……140H～15AH番地の内容を200H番地以降に移す
-d200
1629:0200  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO  移動している
1629:0210  50 51 52 53 54 55 56 57-58 59 5A 00 00 00 00 00   PQRSTUVWXYZ.....
1629:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1629:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1629:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1629:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1629:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1629:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-d100 19f
1629:0100  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
1629:0110  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
1629:0120  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
1629:0130  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
1629:0140  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
1629:0150  50 51 52 53 54 55 56 57-58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[¥]^_  移動元もそのまま
1629:0160  60 61 62 63 64 65 66 67-68 69 6A 6B 6C 6D 6E 6F   `abcdefghijklmno  残っている
1629:0170  70 71 72 73 74 75 76 77-78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
1629:0180  80 81 82 83 84 85 86 87-88 89 8A 8B 8C 8D 8E 8F   ................
1629:0190  90 91 92 93 94 95 96 97-98 99 9A 9B 9C 9D 9E 9F   ................
■
```

Name

機能　FCB(ファイル・コントロールブロック)にファイル名をセットします．

書式　N〈ファイル名 1〉[〈ファイル名 2〉……]

解　説

①N〈ファイル名 1〉

ファイル名を指定しないで DEBUG を起動したときなど，Load コマンドや Write コマンドで使用するファイル名を FCB にセットします．

```
A>debug
-ncommand.com ……Loadコマンドでロードするファイルのファイル名 "COMMAND . COM" をNameコマンドでFCBにセットする
-l……Loadコマンドで "COMMAND . COM" をロード(CS：100H番地からロードされる)
-g……プログラムの実行

Command ﾊﾞｰｼﾞｮﾝ 2.11

A>dir /w

 ドライブ A: のボリュームラベルは PC9800_SYS
 ディレクトリは A:¥

COMMAND  COM    KNJDIC   SYS    CHKDSK   COM    DEBUG    COM    EDLIN    COM
MORE     COM    RECOVER  COM    SYS      COM    CREF     EXE    EXE2BIN  EXE
FC       EXE    FIND     EXE    LIB      EXE    LINK     EXE    MASM     EXE
SORT     EXE    ASSIGN   COM    COPY2    COM    COPYA    COM    DICM     COM
DISKCOPY COM    DUMP     COM    FORMAT   COM    KEY      COM    PRINT    COM
SPEED    COM    SWITCH   COM    USKCGM   COM    COPYA    DOC    README   DOC
COMMAND  DOC    CONFIG   SYS    AUTOEXEC BAT    HANOI10  TXT    C
FACT2000 TXT    FACT3000 TXT    ASM
       38 個のファイルがあります
   325632 バイトが使用可能です

A>exit

Program terminated normally
-■
```

② N 〈ファイル名 1〉［〈ファイル名 2〉 ……］

デバッグされるプログラムに，パラメータとして他のファイルを必要とするときに，その(パラメータとなるファイルの)ファイル名を指定(FCB にセット)します．この指定を行った後に Go コマンドを実行すると，コマンドレベルから実行するコマンド名とパラメータを入力したかのようにそのプログラムが実行されます．ただし，Name コマンドによってパラメータとなるファイル名を指定した後 Write コマンドを実行してしまうと，今デバッグ中のファイルがそのパラメータとして指定したファイルに書き込まれてしまいます．ですから，つねに Load コマンドか Write コマンドを実行する前に，かならず Name コマンドを実行しなければなりません．

```
A>dir

 ドライブ A: のボリュームラベルは PC-9801
 ディレクトリは A:¥bin¥work

.            <DIR>      84-03-23   8:17
..           <DIR>      84-03-23   8:17
MASM     EXE    80856   83-10-14  16:58
FUNC-00  ASM      768   84-02-20  21:54
       4 個のファイルがあります
   693248 バイトが使用可能です
                ┌DEBUGを起動する(起動時にDEBUG MASM . EXE FUNC-00, , FUNC-00 ; と入力すると以下の3つのコマ
A>debug ────────┘ンドを実行したのと同じ結果が得られる)
-nmasm.exe……ロードするファイルのファイル名 "MASM . EXE" をNameコマンドでFCBにセットする
-l……Loadコマンドで "MASM . EXE" をメモリにロードする
-nfunc-00,,func-00;……MASMに渡すパラメータをNameコマンドでセットする．DOSのコマンドレベルでMASMを実行するときと同じようにパラメータを入力する
-g……実行
The Microsoft MACRO Assembler , Version 1.25
 Copyright (C) Microsoft Corp 1981,82,83
```

```
Warning Severe
Errors  Errors
0       0

Program terminated normally……正常に終了した
-q……DEBUGを抜けDOSのコマンドレベルに戻る

A>dir func-00

 ドライブ A: のボリュームラベルは PC-9801
 ディレクトリは A:¥bin¥work

FUNC-00  ASM       768  84-02-20  21:54
FUNC-00  OBJ       116  84-03-23   8:20  } 今アセンブルしたオブジェクトファイルとリストファイル
FUNC-00  LST      1944  84-03-23   8:20  }
        3 個のファイルがあります
   690176 バイトが使用可能です

A>■
```

Name コマンドは以下に示すアドレスのメモリ［PSP(プログラム・セグメント・プリフィックス)の一部］に影響を与えます.

CS：5C　〈ファイル1〉用の FCB
CS：6C　〈ファイル2〉用の FCB
CS：80　文字数(先頭の N を除く)
CS：81　入力されたすべての文字

(参考：Write, Load)

Output

機能　〈値〉で指定されたアドレスのI/Oポートに〈バイト〉で指定された値を出力します.
書式　O〈値〉〈バイト〉

解　説

〈値〉には，16ビットのポート・アドレスを指定します.

```
-oa1 03……アドレスA1Hのポートに03Hを出力する
-o32ae 06……アドレス32AEHのポートに06Hを出力する
-■
```

Quit

機能　DEBUG を終了します．
書式　Q

解　説

Quit コマンドは，DEBUG を終了してコマンドレベルに戻ります．その際，現在デバッグ中のファイルはセーブされませんから，ファイルをセーブしてから終了したいときは，まず Name コマンドでファイル名を指定した後，Write コマンドでそのファイルをセーブし，その後に Quit コマンドで DEBUG を終了しなければなりません．

```
-q□……DEBUGを終了する
A> ■……DOSのプロンプトが表示されコマンドレベルに戻る
```

(参考：Name , Write)

Register

機能　レジスタの内容を設定／表示します．
書式　R［〈レジスタ名〉］

解　説

①R

〈レジスタ名〉を省略して R のみを入力すると，現在着目している命令(IP が指し示すアドレスの命令)と，すべてのレジスタの内容を表示します．

```
-r□……Registerコマンドで，現在プログラム実行時用に設定されているレジスタの内容 を表示する
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1626  ES=1626  SS=1626  CS=1626  IP=0100   NV UP DI PL NZ NA PO NC
1626:0100 45              INC     BP
-■
```

フラグの状態

現在 I P の指しているアドレスの内容とそのニーモニック

②R〈レジスタ名〉

次に示す〈レジスタ名〉を指定すると，その内容を 16 進数で表示して入力待ちになり，キーボードから入力された 16 進数が新しいレジスタ値となります．変更の必要がない場合

は，単にリターンキーを押します．

AX，BX，CX，DX，SP，BP，SI，DI，DS，ES，SS，CS，IP または PC

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1626  ES=1626  SS=1626  CS=1626  IP=0100   NV UP DI PL NZ NA PO NC
1626:0100 45            INC     BP
-rax ……Rに続いてレジスタ名を入力するとそのレジスタに現在設定されている内容を表示して入力待ちになる．AXレジスタの内容を変更する
AX 0000 ……現在AXレジスタは0000H
:3e0d ……新しく設定したい値をキーボードから入力する
-rip ……IPを変更する
IP 0100
:1000 } IPを1000Hに設定しておくとその後Goコマンドを〈実行アドレス〉を省略して実行した場合，1000H番地から実行がはじまる
-r
AX=3E0D  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1626  ES=1626  SS=1626  CS=1626  IP=1000   NV UP DI PL NZ NA PO NC
1626:1000 FFBE7209      ???     [BP+0972]                       SS:0972=1689
-■
```

③ RF

〈レジスタ名〉として〝F〟（フラグ）を指定すると，DEBUGは現在のフラグの状態をTable 3.3のような2文字の英文字で表示します．フラグを変更するには，フラグの状態に対応する2文字のコードを入力します．このとき，どのような順序でフラグを入力してもかまいません．新しい値が入力されなかったフラグの状態はもとのままです．また，1つのフラグに2つの状態を入力した場合はエラーになり，エラーになったところまでのフラグの状態は変更されますが，それ以降は変更されません．

Table 3.3　フラグとRegisterコマンドで使われるコード

フラグ名	セット	クリア
Overflow	OV　オーバー	NV　ノンオーバー
Direction	DN　減少	UP　増加
Interrupt	EI　許可	DI　禁止
Sign	NG　負	PL　正
Zero	ZR　ゼロ	NZ　ノンゼロ
Auxiliary Carry	AC　キャリー	NA　ノンキャリー
Parity	PE　偶数	PO　奇数
Carry	CY　キャリー	NC　ノンキャリー

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1626  ES=1626  SS=1626  CS=1626  IP=0100   NV UP DI PL NZ NA PO NC
1626:0100 45            INC     BP
-rf ……フラグの内容を変更する
NV UP DI PL NZ NA PO NC -cyzrpe ……変更したいフラグの名前を英文字コードで入力する．順序は任意．キャリーフラグ，ゼロフラグ，パリティフラグをそれぞれ1に設定する
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1626  ES=1626  SS=1626  CS=1626  IP=0100   NV UP DI PL ZR NA PE CY
1626:0100 45            INC     BP                          ―― 変更されている
-■
```

DEBUG起動直後，セグメント・レジスタはメモリの空き領域の最下位アドレスに，IPは0100Hに，SPはメモリ量に応じてそれぞれセットされ，残りの汎用レジスタおよびフラグはすべてクリアされます．

なお，Registerコマンドで設定したレジスタの値は，GoコマンドおよびTraceコマンドを実行したときに各レジスタに代入される値であって，Registerコマンドでレジスタを書き換えてもその時点のレジスタ類にはまったく影響がありません．たとえばSPの値を変更しても，DEBUGが使用しているスタック・ポインタは変更されません．これは，あるプログラムをGoコマンドまたは，Traceコマンドで実行した際に設定されるSPの値として扱われます．

Search

機能　指定された範囲で指定された〈リスト〉を検索します．

書式　S〈レンジ〉〈リスト〉

解　説

Searchコマンドは，〈レンジ〉で指定された範囲に〈リスト〉と同じバイトデータの並びがあるかどうかを調べ，見つかった場合はそのアドレスをすべて表示します．

```
A>debug command.com
-s100 200 cd 21 ……100Hから200H番地の範囲でCDH，21Hを検索する
163F:01AE
163F:01B4 } CDH，21Hが見つかったアドレスが表示される
163F:01C4
-d100 200
163F:0100  E9 CD 0B 00 00 00 00 00-00 00 00 00 00 00 00 00   iM..............
163F:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
163F:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
163F:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
163F:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
163F:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
163F:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
163F:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
163F:0180  BA 6A 09 3D 02 00 74 18-3D 05 00 74 13 BA B2 08   :j.=..t.=..t.:2.
163F:0190  3D 08 00 74 0B BA A3 08-3D 0B 00 74 03 BA 8D 08   =..t.:#.=..t.:..
163F:01A0  0E 1F E8 38 04 EB 13 8C-C9 8E D1 BC 80 01 CD 21   ..h8.k..I.Q<..M!
163F:01B0  72 CE B4 4D CD 21 2E A3-CF 09 E9 CE 00 FB 8C C8   rN4MM!.#O.iN.{.H
163F:01C0  8E D8 B4 0D CD 21 F7 06-96 09 FF FF 74 03 E9 A6   .X4.M!w.....t.i&
163F:01D0  02 33 ED C6 06 DA 09 00-C6 06 DA 0A 00 83 3E D6   .3mF.Z..F.Z...>V
163F:01E0  09 00 74 06 C7 06 D6 09-FF FF 80 3E CE 09 00 75   ..t.G.V....>N..u
163F:01F0  03 E9 DC 00 F9 CB BA 1F-09 0E 1F E8 DF 03 80 3E   .i¥.yK:....h_..>
163F:0200  D5                                                U
-■
                                   CDH，21Hのあるアドレス
```

Trace

機能　レジスタやフラグ，実行中の命令のニーモニックを表示しながら，1 命令ずつプログラムを実行します．

書式　T［＝〈アドレス〉］［〈値〉］

解　説

①T＝〈アドレス〉〈値〉

プログラムを〝＝〈アドレス〉〟（等号〝＝〟はかならず付ける）で指定されたアドレスから，〈値〉で指定されたステップ数（16 進数）だけ，レジスタやフラグの内容および実行中の命令のニーモニックを，1 ステップずつ表示しながら実行（トレース）します．なお〝＝〈アドレス〉〟のうち，セグメント指定を省略すると IP のみが設定し直されます．

```
A>debug command.com ↵……DEBUGを起動して "COMMAND.COM" をロードする
-t=100 5↵……100H番地（プログラムの先頭）から5ステップ，プログラムを実行

AX=0000  BX=0000  CX=4326  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD0   NV UP DI PL NZ NA PO NC
163F:0CD0 BC8001        MOV     SP,0180 ――次に実行する命令

AX=0000  BX=0000  CX=4326  DX=0000  SP=0180  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD3   NV UP DI PL NZ NA PO NC
163F:0CD3 B8DF0C        MOV     AX,0CDF

AX=0CDF  BX=0000  CX=4326  DX=0000  SP=0180  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD6   NV UP DI PL NZ NA PO NG
163F:0CD6 B104          MOV     CL,04

AX=0CDF  BX=0000  CX=4304  DX=0000  SP=0180  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD8   NV UP DI PL NZ NA PO NC
163F:0CD8 D3E8          SHR     AX,CL

AX=00CD  BX=0000  CX=4304  DX=0000  SP=0180  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CDA   NV UP DI PL NZ NA PO CY
163F:0CDA 8CC9          MOV     CX,CS
-■
```

レジスタの内容を表示しながら実行する（Registerコマンドが出力するものと同内容）

②T〈値〉

〝＝〈アドレス〉〟を省略すると，現在の CS レジスタと IP の値で指定されるアドレス（Register コマンドで表示される CS レジスタ，IP の値）から〈値〉で指定されたステップ数だけトレースします．通常（DEBUG 起動時や Load コマンドでのファイルのロード時）はプログラムの先頭に CS レジスタおよび IP が設定されており，〝＝〈アドレス〉〟を省略するとプログラムの先頭からトレースすることができます．

```
A>dir tree↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

TREE     EXE    14888  84-03-18  15:30
       1 個のファイルがあります
   701440 バイトが使用可能です

A>debug tree.exe↵……DEBUGを起動して"TREE.EXE"をロードする
-t2↵……IPの指すアドレス(プログラムの先頭)から2ステップトレース

AX=0000  BX=0000  CX=3828  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=19DC  CS=194C  IP=0000   NV UP DI PL NZ NA PO NC
194C:0000 06             PUSH    ES

AX=0000  BX=0000  CX=3828  DX=0000  SP=01FE  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=19DC  CS=194C  IP=0001   NV UP DI PL NZ NA PO NC
194C:0001 BA5219          MOV     DX,1952
-t3↵……続きを3ステップトレース(順にトレースするのであればアドレス指定は不要)

AX=0000  BX=0000  CX=3828  DX=1952  SP=01FE  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=19DC  CS=194C  IP=0004   NV UP DI PL NZ NA PO NC
194C:0004 8EDA            MOV     DS,DX

AX=0000  BX=0000  CX=3828  DX=1952  SP=01FE  BP=0000  SI=0000  DI=4000
DS=1952  ES=163F  SS=19DC  CS=194C  IP=000B   NV UP DI PL NZ NA PO NC
194C:000B 8EC2            MOV     ES,DX

AX=0000  BX=163F  CX=3828  DX=1952  SP=0200  BP=0000  SI=0000  DI=4000
DS=1952  ES=1952  SS=19DC  CS=194C  IP=000E   NV UP DI PL NZ NA PO NC
194C:000E 8BEF            MOV     BP,DI
-■
```

③T＝〈アドレス〉

〈値〉を省略すると，〈アドレス〉から１ステップだけトレースします．

```
A>debug edlin.com test.txt↵
-t=100↵……100H番地から1ステップ，トレースする．ステップ数を指定しないと1ステップ，トレースする

AX=0000  BX=0000  CX=23A4  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0F12   NV UP DI PL NZ NA PO NC
163F:0F12 C6061C2400    MOV     BYTE PTR [241C],00                DS:241C=00
-g↵……続きを実行．GoコマンドとTraceコマンドと組み合せて必要なところだけをトレースするようにすると効率的にデバッグができる
新しいファイルです……EDLINが実行された
*i↵
     1:*Now EDLIN.COM comes up.↵
     2:*^Z↵
*e↵

Program terminated normally
-q↵

A>dir test↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥bin

TEST     TXT       26  84-03-23  10:04……"TEST.TXT"ができている
       1 個のファイルがあります
   700416 バイトが使用可能です

A>■
```

④T

パラメータをすべて省略して〝T〟のみを入力すると，現在のCSレジスタとIPの値で指定されるアドレスから1ステップだけトレースします．

```
A>debug command.com↵……DEBUGを起動して，その管理下に〝COMMAND．COM〟をロードする
-t↵……プログラムの先頭から1ステップ，トレースする(DEBUG起動時はプログラムの先頭にIP，CSレジスタが設定されている)

AX=0000  BX=0000  CX=4326  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD0   NV UP DI PL NZ NA PO NC
163F:0CD0 BC8001        MOV     SP,0180
-t↵……次を1ステップ，トレースする

AX=0000  BX=0000  CX=4326  DX=0000  SP=0180  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD3   NV UP DI PL NZ NA PO NC
163F:0CD3 B8DF0C        MOV     AX,0CDF
-t↵

AX=0CDF  BX=0000  CX=4326  DX=0000  SP=0180  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD6   NV UP DI PL NZ NA PO NC
163F:0CD6 B104          MOV     CL,04
-t↵

AX=0CDF  BX=0000  CX=4304  DX=0000  SP=0180  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=163F  CS=163F  IP=0CD8   NV UP DI PL NZ NA PO NC
163F:0CD8 D3E8          SHR     AX,CL
-■……4ステップ，トレースした
```

Traceコマンドは8086(8088)CPUのハードウェア・トレースモードを使用しています(トレースフラグをセットしているので，ROMの中の命令もトレースすることができます)．

(参考：Go，Register)

Unassemble

機能　〈レンジ〉で指定された範囲のメモリの内容を逆アセンブルして表示します．

書式　U［〈レンジ〉］

解　説

①U〈レンジ〉

〈レンジ〉で指定された範囲のメモリの内容を逆アセンブルします．

```
A>debug format.com↵……DEBUGを起動して，その管理下に〝FORMAT．COM〟をロードする
-u129 140↵……129H～140H番地のメモリの内容を逆アセンブルする
163F:0129 81062A160010  ADD     WORD PTR [162A],1000
163F:012F B419          MOV     AH,19
163F:0131 CD21          INT     21
163F:0133 A24009        MOV     [0940],AL
163F:0136 A26109        MOV     [0961],AL
163F:0139 BE5C00        MOV     SI,005C
163F:013C AC            LODSB
163F:013D 0AC0          OR      AL,AL
163F:013F 7505          JNZ     0146
```

```
-u100 l10↵……100H番地から長さ10H/バイトの範囲を逆アセンブルする
163F:0100 EB09          JMP     010B
163F:0102 56            PUSH    SI
163F:0103 65            DB      65
163F:0104 7273          JB      0179
163F:0106 2032          AND     [BP+SI],DH
163F:0108 2E            CS:
163F:0109 3030          XOR     [BX+SI],DH
163F:010B BC160C        MOV     SP,0C16
163F:010E 50            PUSH    AX
163F:010F BA320C        MOV     DX,0C32
-■
```

②U

〈レンジ〉を省略して"U"のみを入力すると，直前に実行したUnassembleコマンドの終了アドレスの次の命令から(DEBUG起動時，またはLoadコマンドでファイルをロードした場合CS：100Hから，Registerコマンドを実行した場合CS：IPから)20H(32)バイト逆アセンブルします．

```
A>debug exe2bin.exe↵……DEBUGを起動してその管理下に"EXE2BIN.EXE"をロードする
-u↵……プログラムの先頭から20H/バイト逆アセンブルする(DEBUG起動時はプログラム先頭にIPの値がセット されている)
164F:0000 EB09          JMP     000B
164F:0002 56            PUSH    SI
164F:0003 65            DB      65
164F:0004 7273          JB      0079
164F:0006 2032          AND     [BP+SI],DH
164F:0008 2E            CS:
164F:0009 3030          XOR     [BX+SI],DH
164F:000B BC160C        MOV     SP,0C16
164F:000E 50            PUSH    AX
164F:000F BA320C        MOV     DX,0C32
-u↵……次を20H/バイト逆アセンブルする
164F:0020 BAD500        MOV     DX,00D5
164F:0023 B409          MOV     AH,09
164F:0025 CD21          INT     21
164F:0027 1F            POP     DS
164F:0028 CB            RETF
164F:0029 BA7916        MOV     DX,1679
164F:002C 8EDA          MOV     DS,DX
164F:002E E89900        CALL    00CA
164F:0031 7305          JNB     0038
164F:0033 8EDA          MOV     DS,DX
164F:0035 E9B100        JMP     00E9
164F:0038 BF3000        MOV     DI,0030
164F:003B AC            LODSB
-r↵
AX=0000  BX=0000  CX=0478  DX=0000  SP=0080  BP=0000  SI=0000  DI=0000
DS=163F  ES=163F  SS=1697  CS=164F  IP=0000   NV UP DI PL NZ NA PO NC
164F:0000 EB09          JMP     000B
-u↵……Registerコマンドを実行後，Unassembleコマンドを実行するとIPの値のアドレスから逆アセンブルする
164F:0000 EB09          JMP     000B
164F:0002 56            PUSH    SI
164F:0003 65            DB      65
164F:0004 7273          JB      0079
164F:0006 2032          AND     [BP+SI],DH
164F:0008 2E            CS:
164F:0009 3030          XOR     [BP+SI],DH
164F:000B 50            PUSH    AX
164F:000C B430          MOV     AH,30
164F:000E CD21          INT     21
164F:0010 3C02          CMP     AL,02
164F:0012 7D0F          JGE     0023
164F:0014 1E            PUSH    DS
164F:0015 BA7916        MOV     DX,1679
164F:0018 8EDA          MOV     DS,DX
-■
```

③U〈開始アドレス〉

〈レンジ〉のうち，〈開始アドレス〉のみを指定すると，そのアドレスから20Hバイト逆アセンブルします．

```
A>debug exe2bin.exe ……DEBUGを起動して，その管理下に "EXE2BIN.EXE" をロードする
-u1000 ……1000H番地から20Hバイト逆アセンブルする
164F:1000 5A              POP     DX
164F:1001 8B7E04          MOV     DI,[BP+04]
164F:1004 83EC10          SUB     SP,+10
164F:1007 C746F80000      MOV     WORD PTR [BP-08],0000
164F:100C 2BC0            SUB     AX,AX
164F:100E 99              CWD
164F:100F 8946F2          MOV     [BP-0E],AX
164F:1012 8956F0          MOV     [BP-10],DX
164F:1015 C646EC80        MOV     BYTE PTR [BP-14],80
164F:1019 8A5502          MOV     DL,[DI+02]
164F:101C 8856F4          MOV     [BP-0C],DL
164F:101F 807EF400        CMP     BYTE PTR [BP-0C],00
-■
```

Write

機能　指定されたメモリ領域の内容をディスクに書き込みます．

書式　W［〈アドレス〉［〈ドライブ〉〈レコード〉〈値〉］］

解　説

①W［〈アドレス〉］

"W" あるいは "W〈アドレス〉" を指定した場合，メモリの内容がファイルとしてディスクに書き込まれます．その際，書き込むバイト数がBX：CXにセットされ，ファイル名がDEBUG起動時あるいはNameコマンドで指定(ファイル名がFCBにセット)されていなければなりません．ディスクに書き込まれるメモリの領域は，〈アドレス〉を指定するとそのアドレスから，省略するとCS：100HからBX：CXバイトです．

ファイル名を指定してDEBUGにはいったとき，またはLoadコマンドでファイルをロードした場合にはBX：CXにはそのバイト数が，FCBにはそのファイル名がセットされていますから，これらの値を変更しない限り改めて設定する必要はありません．

```
A>command.

Command ﾊﾞｰｼﾞｮﾝ 2.11 ……"COMMAND.COM" 起動時に表示される

A>debug command.com ……DEBUGを起動してその管理下に "COMMAND.COM" をロードする
-d10c0
18A0:10C0  43 4F 4D 53 50 45 43 3D-0D 0A 43 6F 6D 6D 61 6E   COMSPEC=..Comman
18A0:10D0  64 20 CA DE B0 BC DE AE-DD 20 32 2E 31 31 0D 0A   d J^0<^.] 2.11..
18A0:10E0  24 43 4F 4D 4D 41 4E 44-2E 43 4F 4D 20 82 AA 8C   $COMMAND.COM .*.
18A0:10F0  A9 82 C2 82 A9 82 E8 82-DC 82 B9 82 F1 0D 0A 24   ).B.).h.¥.9.q..$
18A0:1100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
18A0:1110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
18A0:1120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
18A0:1130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
-e10d2 'VERSION'↵……10D2H番地以降に"VERSION"と書き込む
-w↵……ファイルに書き込む
Writing 4326 bytes
-q↵……DEBUGを抜ける

A>command↵……今DEBUGで変更を加えた "COMMAND.COM" を実行してみる

Command VERSION 2.11
          └書き換わっている
A>exit↵

A>exit↵

A>debug↵……DEBUGで簡単なプログラムを作ってみる．メモリのプログラムをそのままWコマンドでファイルに書き込めばCOMファイルができてしまう
-a↵……Assembleコマンドでプログラムを入力する
161D:0100 mov ah,09↵
161D:0102 mov dx,0109↵
161D:0105 int 21↵
161D:0107 int 20↵
161D:0109 db 'MS-DOS Hand Book',0d,0a↵
161D:011B db ''Debug.com' Write command sample program $'↵
161D:0145
-rcx↵……CXレジスタにファイルを書き込むバイト数をセットする
CX 0000
:45↵
-ntest.com↵……FCBにファイル名をセットする
-w↵……今作ったプログラムを100H番地から45Hバイト新しいファイル "TEST.COM" に書き込む
Writing 0045 bytes
-q↵

A>test↵……今作ったプログラムを実行する
MS-DOS Hand Book
'Debug.com' Write command sample program
A>■
```

②W〈アドレス〉〈ドライブ〉〈レコード〉〈値〉

すべてのパラメータを指定した場合，ディスクのアブソリュートセクタに書込みが行われます．指定されたメモリの〈アドレス〉から，指定された〈ドライブ〉のディスクに，〈レコード〉で指定された論理レコード番号から，〈値〉で指定されたセクタ数だけのメモリの内容を書き込みます．アブソリュートセクタへの書込みは，ファイル・ハンドルを通さないで行われるので，不用意に書き込むとディスクの内容が壊されてしまう場合があり非常に危険です．

なお，〈値〉は書き込まれるメモリ領域の大きさが，64Kバイト以下になるように設定しなければ正しくセーブされません．この〈値〉の最大値は，ディスクの種類(セクタサイズ)によって異なります．また，〈ドライブ〉は番号で指定しますが，ドライブ"A："，"B："……と番号との対応はシステムによって異なります．

```
A>dir↵

 ドライブ A: のボリュームラベルは Hand Book
 ディレクトリは A:¥

COMMAND  COM     17190  84-03-23  12:04
KNJDIC   SYS     56320  83-09-20  14:22
BIN          <DIR>      84-02-25  14:05
COM          <DIR>      84-02-25  14:05
AUTOEXEC BAT        46  84-02-23  17:48
USR          <DIR>      84-02-25  14:58
        6 個のファイルがあります
   712704 バイトが使用可能です
```

```
A>debug↵ ルート・ディレクトリをメモリの0H番地以降に1セクタ読み込む
-l0 0 5 1 (この場合8インチ2Dなので，ルート・ディレクトリはレコ ┌ファイル属性(隠されたファイル)
-d0          ード5以降にある)
17BC:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00   IO      SYS'....
17BC:0010  00 00 00 00 00 00 54 4C-5F 07 02 00 00 A0 00 00   ......TL_....  ..
17BC:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00   MSDOS   SYS'....
17BC:0030  00 00 00 00 00 00 40 72-33 07 2A 00 00 45 00 00   ......@r3.*..E..  ルート・
17BC:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00   COMMAND COM ....  ディレクトリ
17BC:0050  00 00 00 00 00 00 82 60-77 08 43 00 26 43 00 00   .......`w.C.&C..
17BC:0060  48 61 6E 64 20 42 6F 6F-6B 20 20 08 00 00 00 00   Hand Book  .....
17BC:0070  00 00 00 00 00 00 63 70-59 08 00 00 00 00 00 00   ......cpY.......
-eb 20↵ ｝"IO.SYS" と "MSDOS.SYS" のファイル属性を   └ボリュームラベルの属性   ボリュームラベル
-e2b 20↵｝一般のファイルと同じにする
-e60 'ASCII Corp.'↵……ボリュームラベルを "ASCII Corp." にする
-d0↵……メモリにうまく書き込まれたかどうか見てみる
17BC:0000  49 4F 20 20 20 20 20 20-53 59 53 20 00 00 00 00   IO      SYS ....
17BC:0010  00 00 00 00 00 00 54 4C-5F 07 02 00 00 A0 00 00   ......TL_.... ..
17BC:0020  4D 53 44 4F 53 20 20 20-53 59 53 20 00 00 00 00   MSDOS   SYS ....
17BC:0030  00 00 00 00 00 00 40 72-33 07 2A 00 00 45 00 00   ......@r3.*..E..
17BC:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00   COMMAND COM ....
17BC:0050  00 00 00 00 00 00 82 60-77 08 43 00 26 43 00 00   .......`w.C.&C..
17BC:0060  41 53 43 49 49 20 43 6F-72 70 2E 08 00 00 00 00   ASCII Corp......
17BC:0070  00 00 00 00 00 00 63 70-59 08 00 00 00 00 00 00   ......cpY.......
-w0 0 5 1  今修正した部分をディスクに書き込む
-q↵
    開始アドレス  ドライブA  レコード番号  セクタ数
A>^C バッファをクリアするため，CTRL－Cを押す(こうしないとDIRコマンドを入力しても修正前のディレクトリが表示される場合がある)

A>dir↵

 ドライブ A: のボリュームラベルは ASCII Corp.
 ディレクトリは A:¥              └ボリュームラベルが変更されている

IO       SYS     40960  83-10-31    9:34 ｝"IO.SYS" と "MSDOS.SYS" がDIRコマンドで表示されるようになった
MSDOS    SYS     17664  83-09-19   14:18 ｝
COMMAND  COM     17190  84-03-23   12:04
KNJDIC   SYS     56320  83-09-20   14:22
BIN          <DIR>      84-02-25   14:05
COM          <DIR>      84-02-25   14:05
AUTOEXEC BAT        46  84-02-23   17:48
USR          <DIR>      84-02-25   14:58
       8 個のファイルがあります
  712704 バイトが使用可能です

A>■
```

(参考：Load，Name)

第III部
MS-DOS の内部構造

1章 MS-DOSのソフトウェア体系

本章では，MS-DOSを構成する各種ソフトウェアの機能とその相関関係を述べるとともに，MS-DOSのすべての制御を行うモジュールが，どのように起動され，メモリ上に配置されていくかを順を追って解説していきます．

なお，各モジュールの詳細な解説は2章以降で順次取り上げていくことにします．

1.1 MS-DOSの基本構成

MS-DOSをコマンドレベルで利用している限り，システムとしてのMS-DOSを意識することはあまりありません．むしろ意識する必要のないものこそ優れたDOSといえるのでしょう．MS-DOSもそういった点を考慮してか，〝MSDOS．SYS〟，〝IO．SYS〟といった制御モジュールは，隠れたファイルとしてユーザーの目には触れないようになっています．

しかし，アセンブラでのソフトウェア開発や，周辺装置を拡張しようとする場合には，MS-DOSを構成する各モジュールの機能と関係を理解する必要があります．

本節では，MS-DOSの内部構造を知る手始めとして，〝MSDOS．SYS〟，〝IO．SYS〟，〝COMMAND．COM〟の3つのモジュールの概要と相関関係について簡単に説明しています．

1.1.1 COMMAND．COM(コマンド実行部)

〝COMMAND．COM〟は，ユーザーが入力したコマンドを解釈，実行するという機能があります．入力されたコマンドが内部コマンドであれば，〝COMMAND．COM〟の各コマンド処理ルーチンへ移り，外部コマンドであれば，ディスクからロードして実行します．

1.1.2 MSDOS．SYS(ファイル管理部)

〝MSDOS．SYS〟は，MS-DOSの中核を成す部分であり，ファイル，入出力，システムコールの管理を行っています．

MS-DOSでサポートしている入出力装置としては，キーボード，スクリーン，プリンタ，補助入出力，ディスク・ドライブ(5インチ／8インチ フロッピーディスク，ハードディスク)，マウス，リアルタイムクロックなどが挙げられますが，〝MSDOS．SYS〟では，これらの周辺装置をいくつかに分類して，似たような使用が可能なものは，できるだけ同一の手順で扱えるようにしています(たとえばスクリーンとプリンタなど)．また，ディスク上のファイル，デバイスファイ

ルなどのファイル処理，メモリ管理，プロセス管理(プログラムのロード，終了コードの扱い)なども〝MSDOS．SYS〟の機能です．

1.1.3　IO．SYS(入出力制御部)

〝IO．SYS〟は，別名 BIOS(Basic Input Output System)とも呼ばれ，〝MSDOS．SYS〟の入出力要求に応じて実際の入出力操作を行います．

この〝IO．SYS〟に入出力のコール(BIOS コール)を発行するのは〝MSDOS．SYS〟が行い，一般にユーザーが〝IO．SYS〟をコールすることはありません．BIOS コールの形式は，たとえば指定したアドレスに，あるドライブ中のディスクの，あるセクタを転送するといったものとなっています．

〝IO．SYS〟はこの命令を受け取って，FDC や DMAC などにコマンドを発行して，結果(読出し／書込みは正常か否か，エラーの発生またはその種類など)を〝MSDOS．SYS〟に返します．〝IO．SYS〟が標準で扱うデバイスは，ディスク，コンソール，プリンタ，補助入出力，リアルタイムクロックの 5 種類です(Fig. 1.1)．

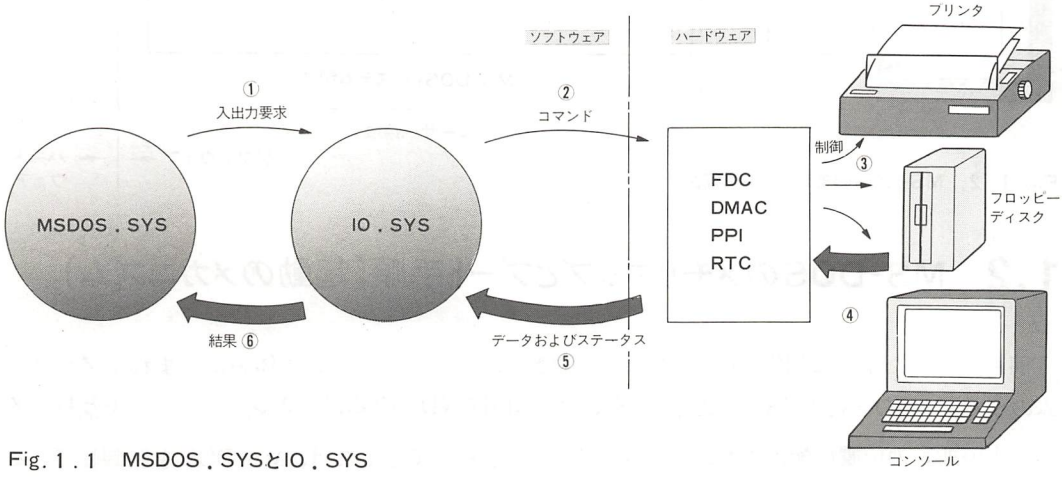


Fig. 1．1　MSDOS．SYSとIO．SYS

1.1.4　各モジュールの関係

今まで述べてきた 3 つのモジュールは，メモリ上に常駐して外部コマンドなどの実行を補助しています．ここで外部コマンドが実行され，コンソールから 1 文字入力を受け付けるような場合を例に，各モジュールの関係を考えてみましょう．

まず，外部コマンドは〝MSDOS．SYS〟へ 1 文字入力を要求します．すると，〝MSDOS．SYS〟はその要求を受け取って，〝IO．SYS〟へ 1 文字入力するように要求します．この後，〝IO．SYS〟

はキーボードから1文字入力して，その文字を"MSDOS．SYS"へ送ります．そして最後に，最初に要求を出した外部コマンドへ目的とする文字が送られ，そのプログラムは処理を続けることになります．Fig. 1.2は，これらのモジュールの相関関係をまとめたものです．

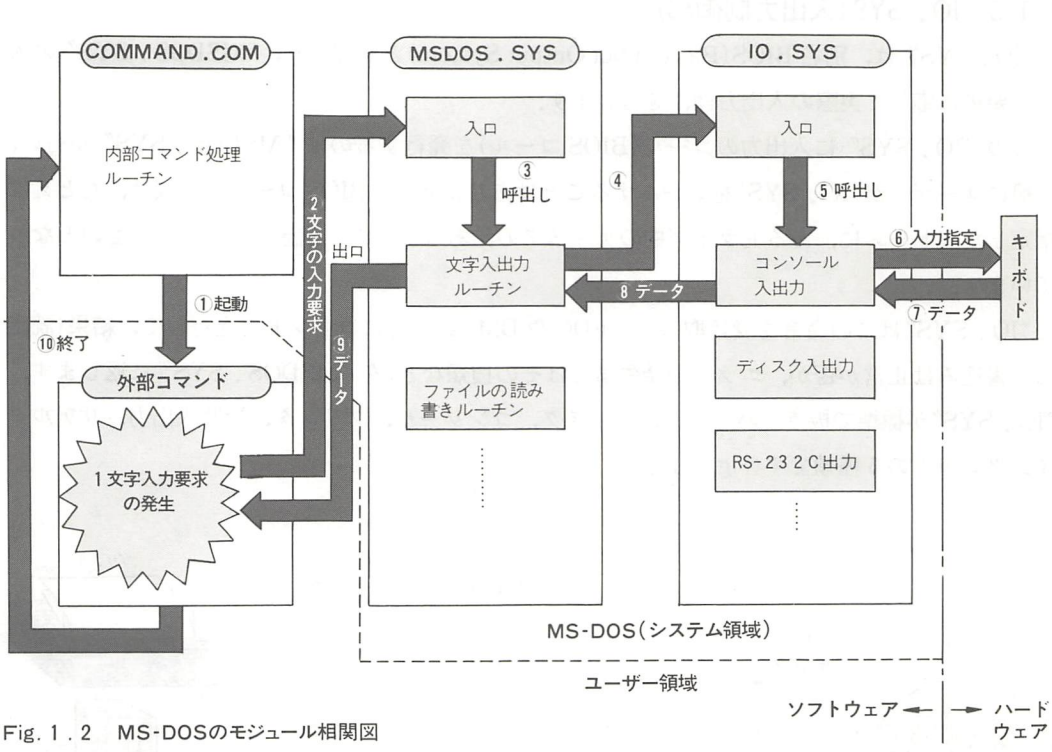


Fig. 1．2　MS-DOSのモジュール相関図

1.2　MS-DOSのメモリマップとブート手順（起動のメカニズム）

前節で述べたようにMS-DOSのすべての働きは，そのソフトウェア体系に含まれるプログラムのうち，"MSDOS．SYS"，"IO．SYS"，"COMMAND．COM"の3つのモジュールとしてメモリ上の所定の位置に配置され，制御されています．そこで本節では，MS-DOSを制御している各プログラムが，メモリ中にどのように配置されているかを，MS-DOSをブートした時点から順を追って見ていくことにします．

1.2.1　ブート時のメモリマップ

まず，システムの電源投入またはリセットの直後のメモリの状態をFig. 1.3に示します．

8086CPUのアーキテクチャでは，割込みベクトルテーブルが，メモリ・アドレスの最下位から置かれているために，だいたいどのシステムでも，このようにRAMがシステムの下位アドレス（セグメント＝0000H）から占めており，メモリの最上位にROMが存在します．

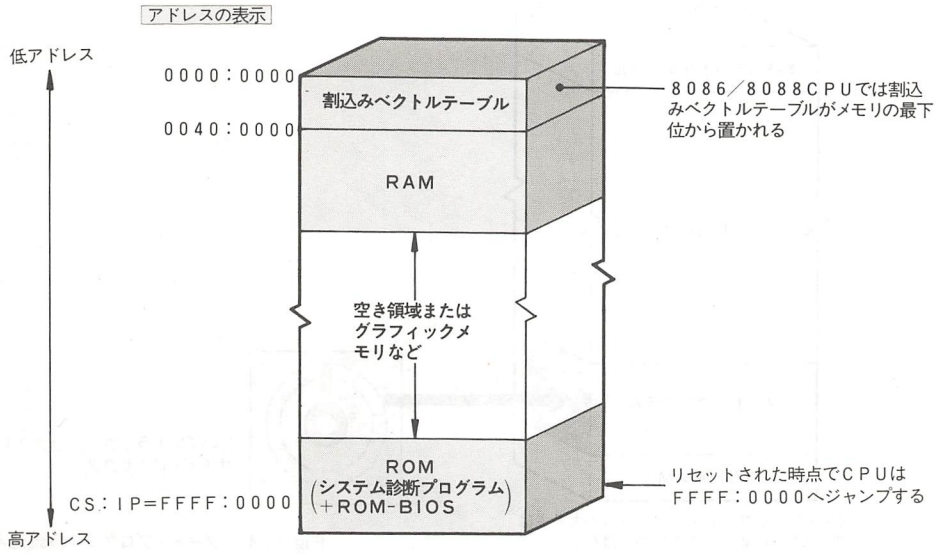


Fig. 1．3　リセット直後のメモリマップ

RAMの最下位の1Kバイトは，8086CPUの割込みベクトルテーブルとなっており，ユーザーが自由に使用できる領域は，セグメント・アドレス0040Hからとなります．

MS-DOSでは，セグメント・アドレス0000HからRAMが存在していないと，DOSを使用することができません．また，途中に一部でもメモリに不連続な領域があると(たとえば，セグメント・アドレス1000Hから64KバイトだけRAMがないような場合)，その不連続領域以降は，MS-DOSの管理対象外となります．

1.2.2　ブート・ストラップ

ROMの中には，通常システムのチェックをするルーチン(システム診断プログラム)やROM-BIOSが記録されています．このBIOSは，MS-DOSだけでなく，CP／Mやほかの OSにも対応できるように作られているため，"IO．SYS"は，実際にFDCやDMAC，クロック・デバイスなどを直接操作するわけではなく，これらROM-BIOSを通して，ディスクやキーボード，プリンタなどを操作します．このようにするメリットは，"IO．SYS"やCP／Mにおける BIOSなどが作りやすく，またDOSのサイズも小さくなることにあります．

さて，リセットの直後にROMの中のプログラムが起動され，まずROM-BIOSのセットアップ(使用開始準備)やハードウェアのチェック(周辺機器が正しく接続されているか，RAMが正しく動作しているかなど)が行われます．

その後，システムディスクからROM-BIOSがブート手続きのプログラムをメモリ上へロードします(Fig. 1.4)．

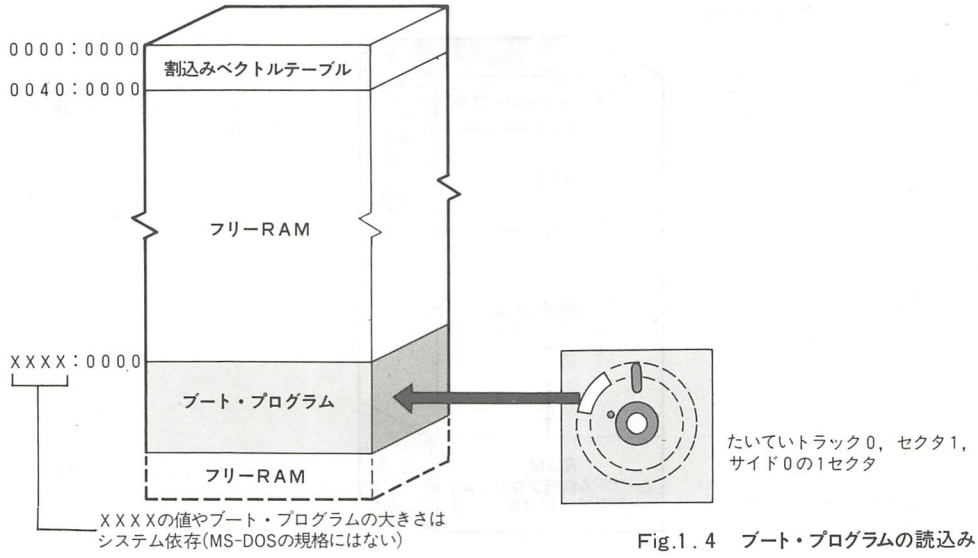
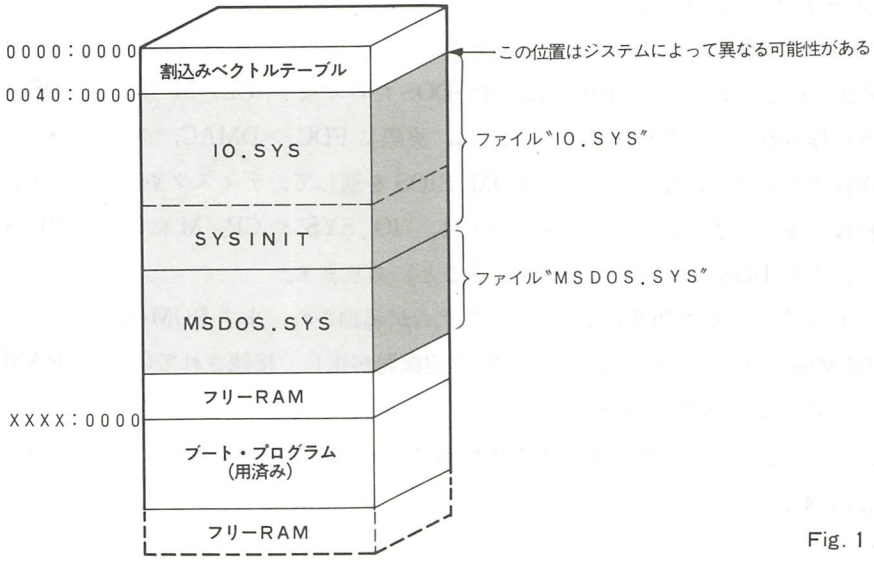


Fig.1．4　ブート・プログラムの読込み

MS-DOSのブート・プログラムは，各ディスクのトラック0，セクタ1，サイド0に書かれていますが，この内容はメーカーにより異なる可能性があるため，異なったメーカーのMS-DOSのシステムディスクではブート動作が正しく行われません．

このブート・プログラムが正しくロードされると，ROM-BIOSからこのプログラムへ制御が移ります(JMPする)．

次にブート・プログラムは，MS-DOSの中心となる3つのファイルのうち，〝IO．SYS〟と〝MSDOS．SYS〟をメモリ上へロードします(Fig. 1.5)．



Fig．1．5　ブート

このうち〝IO．SYS〟は，システム常駐部(IO．SYS)と初期化のみに使われるルーチン(SYSINIT)の2つから成っています．

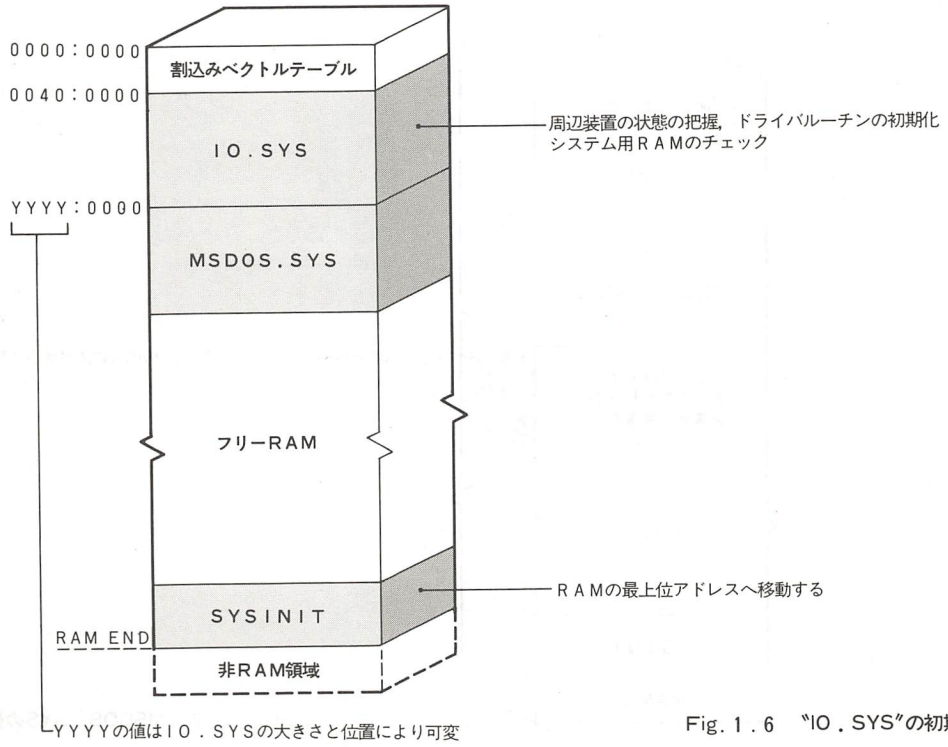
1.2.3　IO．SYSルーチンの初期化動作

ブート・プログラムの処理はFig.1.5に示したメモリマップの状態で終了し，その後の制御は〝IO．SYS〟ルーチンに移っています．

IO．SYSルーチンでは，接続されている周辺装置の状態を把握し，それに合せて各ドライバルーチンの初期化を行います．たとえば，システムに接続されているディスク・ドライブの種類(8インチか5インチか，片面か両面か，ハードディスクかフロッピーディスクかなど)やドライブ数の把握，あるいはキーボード通信用ポート(16ビットマシンでは，キーボード・コントロールに専用のCPUを使っているものが多い)やプリンタ，RS-232Cコントローラの初期設定などが行われます．

また，システムが使用できるRAMの容量を調べるのもこの部分の仕事です．これらのチェックが終了すると，SYSINITルーチンへ制御が移ります．

SYSINITルーチンは，自分自身をメモリの最上位へロードし直し，次にSYSINITルーチンの移動によって空いた分だけ(正確には〝IO．SYS〟の初期化部分で指定された量だけ，通常はSYSINITルーチンの分だけ)，〝MSDOS．SYS〟を移動させます(Fig.1.6)．



Fig．1．6　〝IO．SYS〟の初期化

ここまでくると，"MSDOS．SYS" のメモリ上の位置が定まるため，SYSINIT ルーチンは "MSDOS．SYS" の初期化を行います．具体的には，"MSDOS．SYS" の初期化ルーチンが呼ばれることになります．

1.2.4　MSDOS．SYS の初期化

"MSDOS．SYS" の初期化ルーチンでは，接続されている標準の周辺機器(コンソール，プリンタ，ディスク・ドライブ，クロック・デバイス，通信回線)を管理するための作業領域を確保します．また，ファイルを扱うための管理テーブル(システムテーブル)やディスクとの転送用バッファ領域などの確保も行います．そのほかにも，システムコールをサポートするためのさまざまな初期化を行った後，いったんさきほどの SYSINIT ルーチンへ制御を戻します．

さて，ここまでの一連の初期化動作により，MS-DOS の機能のうちシステムコール(メモリ管理用コール以外)はすべて使用可能となっています．

1.2.5　CONFIG．SYS の設定

SYSINIT ルーチンは，ここでファイル "CONFIG．SYS" を読み込もうとします．このファイル(Configuration File)はオプションであり，なくてもかまいませんが，ユーザーが新しいデバイス・ドライバをシステムに付加したり，DOS で一度に扱うことのできるファイルを増やしたりするときに必要になります．

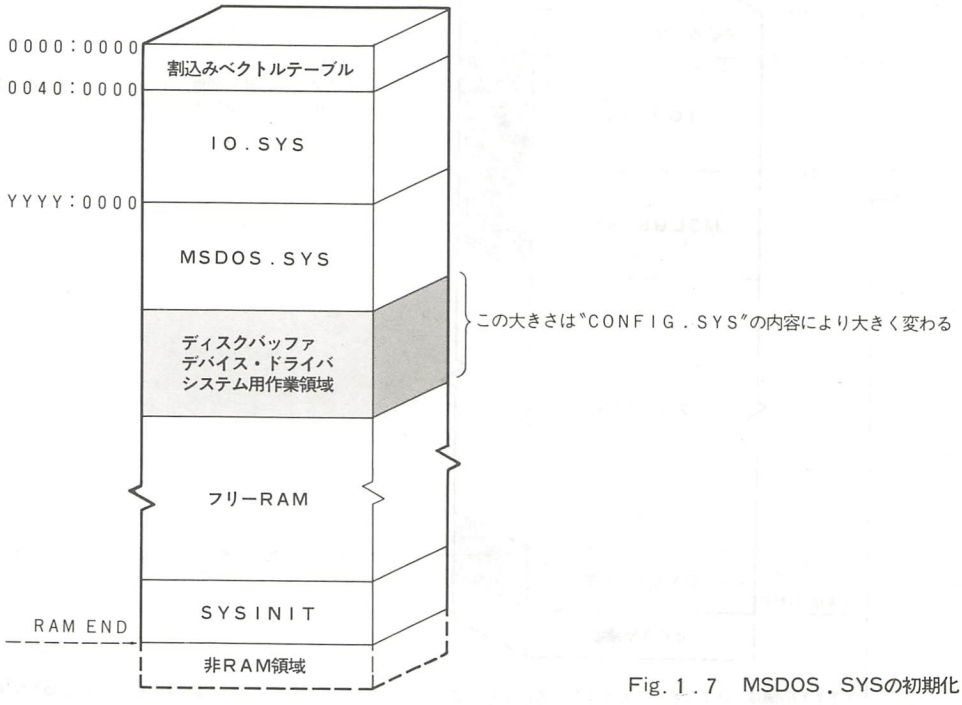


Fig. 1．7　MSDOS．SYSの初期化

この〝CONFIG．SYS〟を読んで解釈，実行するのが，さきほどのSYSINITルーチンなのです．新たにシステムに付加されたデバイス・ドライバは，メモリマップ上では〝MSDOS．SYS〟の次にロードされます．また，〝CONFIG．SYS〟の中でディスクバッファ(ディスクとやりとりするデータを一時的に置いておくための領域)の数を指定することができますが，それらのバッファ領域は，やはりこの〝MSDOS．SYS〟の後ろのメモリ領域内にとられます．

SYSINITルーチンが〝CONFIG．SYS〟を解釈した後のメモリの状態をFig. 1.7に示します．

1.2.6　COMMAND．COMのロード

〝IO．SYS〟，〝MSDOS．SYS〟の初期化動作は，Fig. 1.7の状態ですべて終了しました．後は〝COMMAND．COM〟をメモリ上へロードして制御を移せば，初期化の処理はすべて終了することになります．

〝COMMAND．COM〟も〝IO．SYS〟と同様2つの部分から成っていますが，初期化ルーチンと本体というように分かれているのではなく，常駐部と非常駐部が区別されています．ですから，SYSINITルーチンによりロードされ，〝COMMAND．COM〟に制御が移った後，まず最初に非常駐部が，Fig. 1.8に示すように使用可能なメモリの最上位へディスクから転送されます．

〝COMMAND．COM〟の非常駐部では，ユーザーの入力したコマンドの解釈，実行を行いま

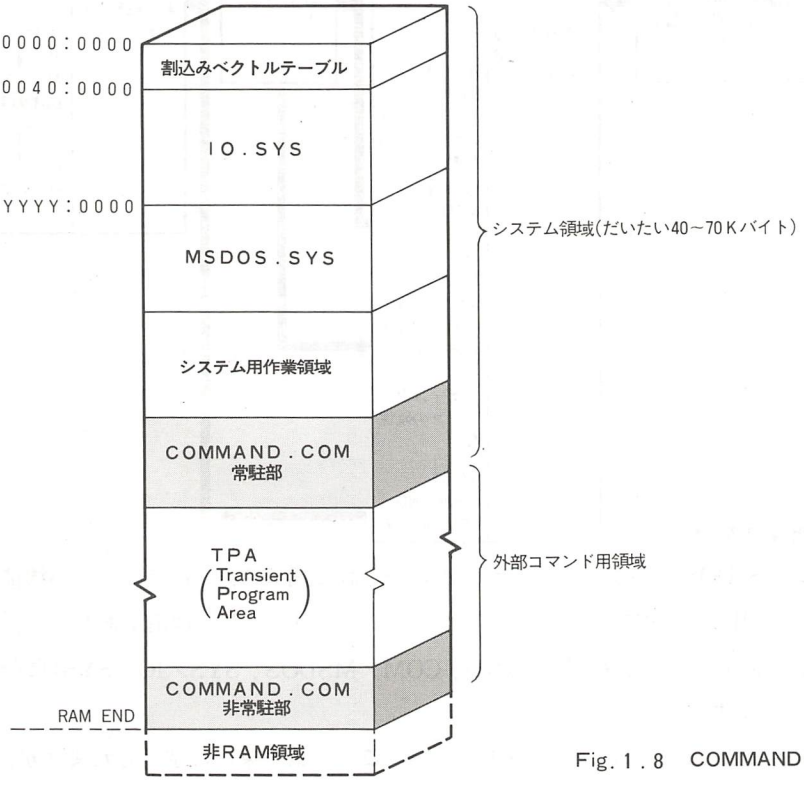


Fig. 1．8　COMMAND．COMのロード

す．DIR や COPY などの内部コマンドは，すべてこの〝COMMAND．COM〟で処理されます．

CHKDSK や DEBUG などの外部コマンドの場合は，これらのプログラムが Fig. 1.8 の TPA のところにロードされ，その外部コマンドへ制御が移ります．このときそのプログラムで使用できる RAM は，ロードされた位置から RAM の最上位までとなります．つまり，〝COMMAND．COM〟の非常駐部を破壊(使用)してもかまわないのです．

外部コマンドの実行が終わった場合，非常駐部は破壊されている可能性があるため，制御は常駐部へ移ります．常駐部では，非常駐部が外部コマンドによって破壊されているか否かを，チェックサムの技法を用いて調べ，必要があれば非常駐部を再度ロードします．もちろん，破壊されていないとわかれば再ロードはしませんので，外部コマンドを実行するたびにロードするよりも〝COMMAND．COM〟の反応が速くなります．

1.2.7　初期化動作の意味

前項で解説した初期化動作はかなり複雑な様相を呈していますが，まとめてみると Fig. 1.9 のようになります．

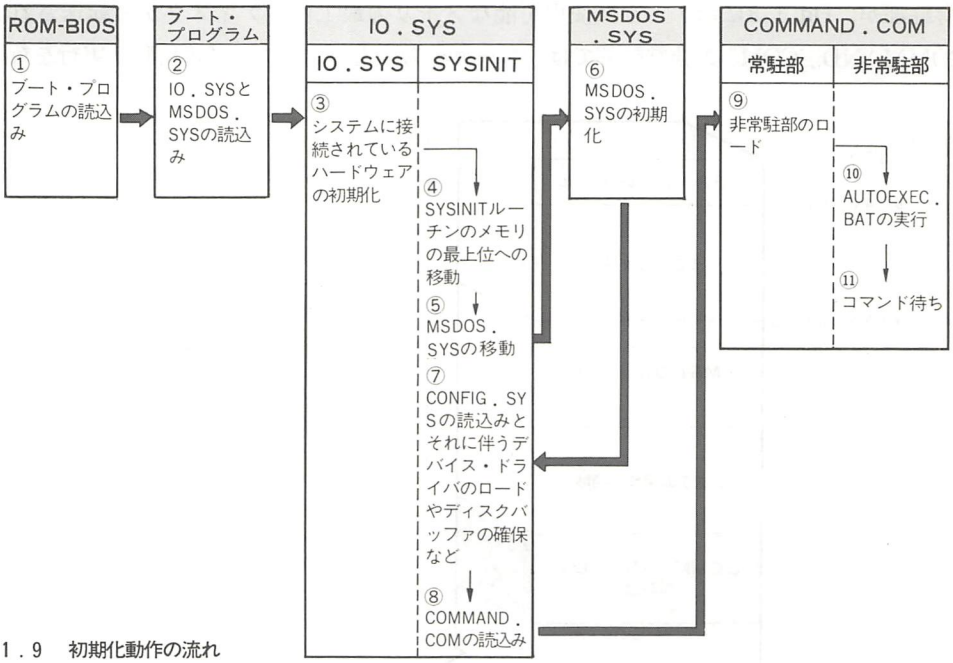


Fig. 1．9　初期化動作の流れ

このように，MS-DOS がいくつかのファイルに分かれて構成され，各ファイルが機能分担を行いながらもさらに相互に複雑な関係を持っているのには，いくつかの理由があります．

まず初めに，なぜ 3 つの部分(COMMAND．COM／MSDOS．SYS／IO．SYS)に分かれているのかを考えてみます．

ある DOS が普及するためには，使いやすいということが第 1 条件に挙げられますが，そのほか

に，多くの種類のシステムで稼動するということも大きな要素です．そのためには，ハードウェアに依存する部分をなるべく少なくする必要があり，DOSがサポートするハードウェアもかなり限定されたものとなります．MS-DOSでは，コンソール(スクリーン＋キーボード)，ディスク，プリンタ，また，もし接続されていればサポートするものとして通信回線，クロック・デバイスの5種類が基本となっています．

これらのハードウェアを操作するデバイス・ドライバは，多分に機種に依存せざるを得ないものなので，これらはまとめて1つのファイルにしてあります．それが〝IO．SYS〟なのです．〝IO．SYS〟では，さきほどの5種類のデバイスしかサポートされませんので，ユーザーが新たに付加したデバイスをサポートすることができません．そのためMS-DOSでは，ユーザーがデバイス・ドライバのルーチンを作ることができるようになっています．しかし，新たに作ったデバイス・ドライバを，そのつど〝IO．SYS〟に組み込んでいたのでは，ユーザーあるいはデバイスをサポートする側で手間がかかるため，〝CONFIG．SYS〟というファイルを用意し，その中にデバイス名を登録しておくだけで(エディタで簡単に登録できる)自動的にシステムがロードするようになっているのです．

次に，機種に依存しない部分について考えてみましょう．

この部分には，システムコールを処理するルーチン(デバイスやファイル，メモリなどの管理)やユーザーが入力したコマンドを解釈，実行するルーチンがあります．このうち前者は〝MSDOS．SYS〟で，後者は〝COMMAND．COM〟で実行されます．〝COMMAND．COM〟が別ファイルになっているのは，主にその機能に起因しますが(MSDOS．SYSとCOMMAND．COMの働きはまったく異なる)，さらに，必要に応じて新しいコマンド・プロセッサと置き換えるため，という理由もあります．

MS-DOSの最新バージョンでは，VISUAL SHELLと呼ばれるコマンド・プロセッサがあり，機能的には〝COMMAND．COM〟とほとんど同じですが，その操作の大半はマウスを使ってインタラクティブ(対話的)に行うことができます．

このように，コマンド・プロセッサを置き換えて使うためにも，〝COMMAND．COM〟は〝MSDOS．SYS〟と分離していなければならないのです．

これまで述べてきたことと，さきほどのブートの手続きを考え合せてみれば，それが妥当なものであることが理解できると思います．

まず最初に，〝IO．SYS〟と〝MSDOS．SYS〟をメモリにロードし，順次初期化動作を行います．次に〝MSDOS．SYS〟のシステムコールを用いてファイル〝CONFIG．SYS〟をオープンし，その内容に従ってデバイス・ドライバをロードします．

その後，SYSINITルーチンは消滅して，代わりに〝COMMAND．COM〟に制御が移ります．〝COMMAND．COM〟の非常駐部はメモリの最上位へ移動して，TPAをできるだけ大きく確保し，その結果常駐部以降はすべてTPAとしてユーザー(のプログラム)が使えることになるのです．

2章 COMMAND.COM

MS-DOSのコマンド・プロッセッサである〝COMMAND.COM〟は，I／Oのリダイレクト，パイプといったUNIXのシェルの概念を取り入れています．本章では〝COMMAND.COM〟の特徴を中心に，その構造について解説していきます．

2.1 コマンドの入力と実行

〝COMMAND.COM〟には，ユーザーが入力したコマンドを実行するという機能があります．たとえば，コンソールにプロンプトマークを出力し，ユーザーが〝DIR〟とタイプしたらカレント・ドライブにはいっているディスクのファイル名一覧をスクリーンに表示して，再び次のコマンド入力を持つ，というような具合いです．また，入力されたコマンド名が，DOSの内部コマンドでなければディスク上からその外部コマンドをロードして実行させます．以下ではユーザーがコマンドラインにコマンドを入力した後，〝COMMAND.COM〟がそれをどのように解釈，実行していくのか，その過程について解説していきます．

2.1.1 ファイルの実行順序

I／Oのリダイレクトやパイプの機能を省いた場合，コマンドラインに入力されるコマンドの一般形は次のようになっています．

〈コマンド名〉[〈パラメータ〉…]

このうち，〝COMMAND.COM〟にとって意味があるのは〈コマンド名〉のみであり，パラメータの列は単に起動されたコマンドに渡されるだけです．

コマンド名としては内部コマンドと外部コマンドがありますが，このうち後者はさらに次の3つに分けられます．

(1) COMファイル
(2) EXEファイル
(3) BATファイル

3つの型のファイルはそれぞれ，ファイル名の拡張子が〝.COM〟，〝.EXE〟，〝.BAT〟となっていることで区別されます．

内部コマンドと外部コマンドは，合せて4種類あるので，ときには同じ名前のコマンドが存在

することが予想されます．つまり，内部コマンドの COPY に対して，"COPY．COM"や"COPY．EXE"，"COPY．BAT"のようなコマンドファイルができる可能性があるのです．このようなとき，４つのコマンドは次のような順序で検索されます．

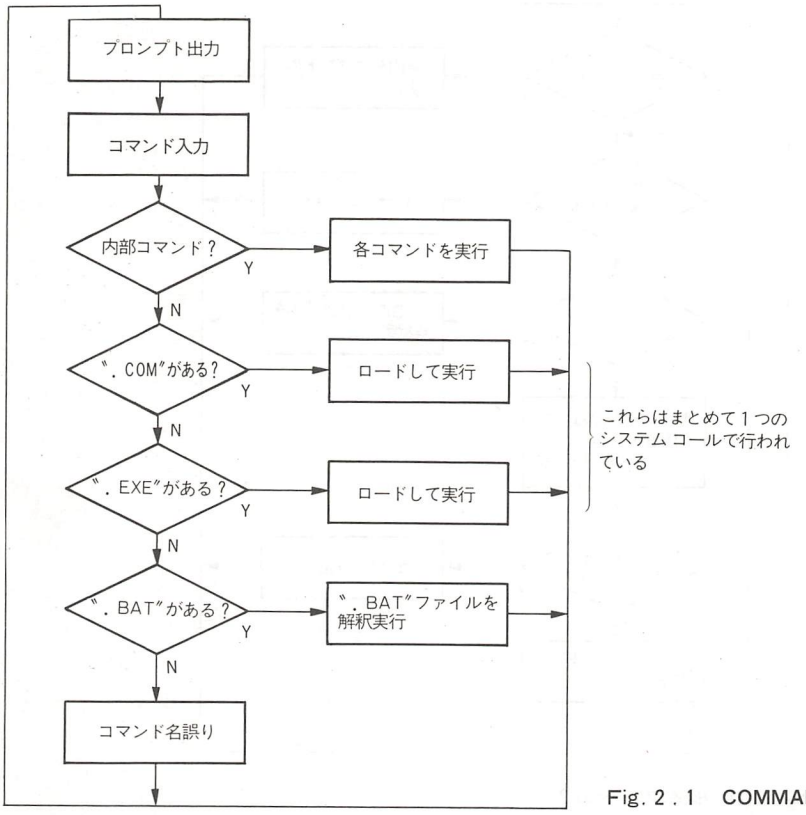
内部コマンド⟶COM ファイル⟶EXE ファイル⟶BAT ファイル

この検索の順序では，つねに内部コマンドが最優先で検索されます．これは，他のコマンドの判定にはディスク・アクセスが伴うために時間がかかることと，"COMMAND．COM"はメモリに常駐しているために，内部コマンドかどうかの判定はすぐできるからです．

EXE ファイルと COM ファイルは，ともに 8086CPU の機械語で記述されたコマンドであるという点では一致しますが，その実行時のメモリの使い方が少々異なります．詳しくは３章を参照してください．

"COMMAND．COM" は，基本的にはユーザーからのコマンドを受けて対応する内部コマンドを実行したり外部コマンドを実行させたりしているだけですので，動作の概略は Fig.2.1 のようになります．

この図はかなり大雑把なフローですが，本節以降の "COMMAND．COM" の解説の中でしだいに厳密化していきます．



Fig．2．1　COMMANDの動作の概略フロー

2.1.2　パスの検索処理

MS-DOSでは，コマンドの検索に対してパス名を指定することができます．たとえば，次のようにコマンドのサーチパスを

```
PATH　A：¥BIN；B：¥BIN
```

としておけば，指定されたコマンド(ファイル)がカレント・ディレクトリにない場合，〝A：¥BIN〞(ディレクトリ名)を捜します．もしこのディレクトリにもない場合，〝B：¥BIN〞の中を捜し，ここにもなければ，コマンドが存在しない旨のエラーメッセージを出力します．

さきほどのフロー(Fig. 2.1)をこのようにするためには，それほど大きな変更は必要ありません．つまり，検索するファイルが指定したディレクトリになかった場合には，設定されているパスを1つずつ切り出してコマンド名の前に付け加え，新しいパス名を作って再度ファイルを検索

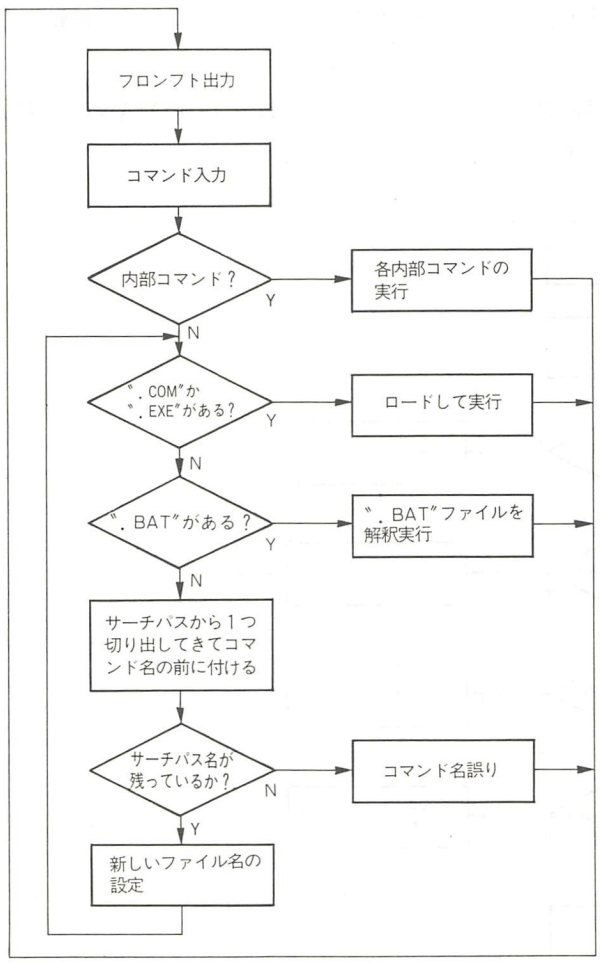


Fig. 2．2　COMMANDの動作の概略フローその2

すればよいのです(ちなみにパス名を与えてそのファイルがあるか否かを調べるには，MS-DOSのシステムコールが使える)．たとえば，コマンド名がXYZの場合，以下のファイルを捜します(Fig. 2.2)．

```
XYZ．*
A：¥BIN¥XYZ．*
B：¥BIN¥XYZ．*        (〝*〞にはCOM，EXE，BATの3つが該当する)
```

2.1.3　バッチコマンドの処理

バッチ処理だからといって，何も特別なものを考える必要はありません．バッチ処理とそうでない普通のコマンド入力では，単に入力方法が異なるだけです．さきほどのフロー(Fig. 2.2)に当てはめて考えれば，〝コマンド入力〞となっている部分が，キーボードからではなくファイルから行われるだけで，後はすべて同じフローになります．

このコマンド入力がどこから行われているかを示すためにCIN(Current Input File)という領域が用意されているとします．この中には，現在の入力がコンソールなら〝CON〞を入れておき，バッチファイルの処理中なら〝TEST．BAT〞のように，現在処理中のバッチファイルの名前を入れておきます．

このCINは，〝COMMAND．COM〞の起動時は〝CON〞となっていますが，バッチファイルの処理中にはその処理中のファイルが割り当てられます．ですから，〝SAMPLE．BAT〞というファイルが処理中であったとすれば，〝CIN←SAMPLE．BAT〞というようにファイル名をCINに代入しておけば，コマンド入力は基本的にファイルとCINから1行取り出してくるだけで済むため，現在の入力が何から行われているか意識しなくてもよくなります．

ただし，バッチコマンドではパラメータや環境変数を処理しなければならないので，コマンド入力部でこれらの置換えが行われます．この処理は，ファイルからコマンド行を1行取り出してきてその内容を調べ，〝%〈数字〉〞ならば〝parameter[数字]〞で置き換え，〝%〈環境変数〉%〞ならばそれを〝environment[環境変数]〞で置き換えます．後はコマンド解析ルーチンへそれを渡せばよいのです．ただし，〝parameter[n]〞は，バッチファイルに対するn番目のパラメータ(の文字列)を意味し，〝environment[環境変数]〞は，〈環境変数〉にセットされている文字列のことを指します(環境はSETコマンドでセットする)．

ですから，バッチファイル中の1行の

```
SAMPLE　%1　%TTY%　ABC
```

のような文字列は，次のような形に置き換えられた後にコマンド解析ルーチンに送られます．

```
SAMPLE　ARG1　VT100　ABC
```

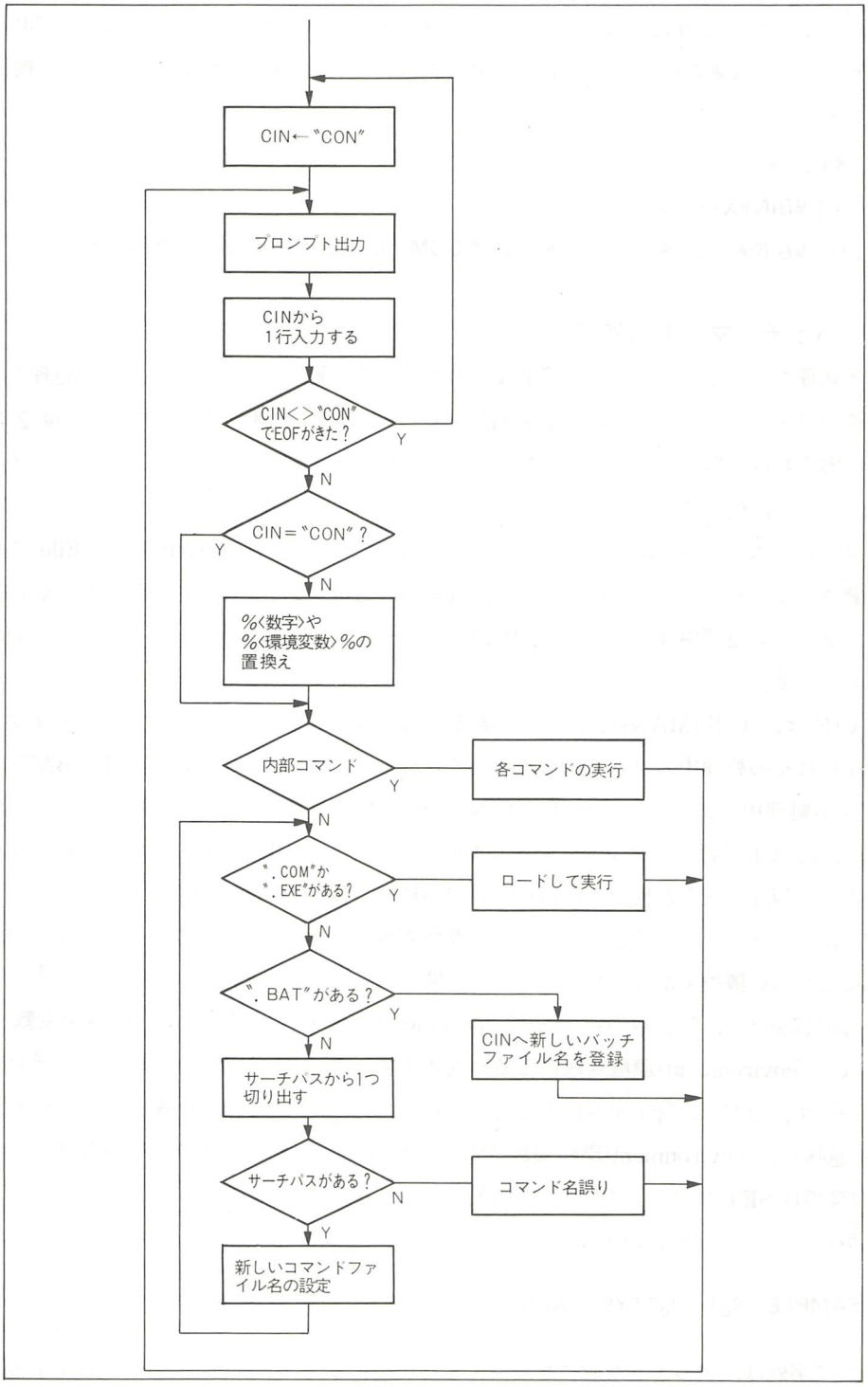


Fig. 2．3　COMMAND動作フロー

なお，環境の環境変数と文字列については〝COMMAND．COM〟とは別のメモリ領域に保存されていますが，バッチファイルのパラメータは〝COMMAND．COM〟自身で正しく管理しておかなければなりません．

さてここまでくれば，さきほどの不完全なフローをどのように修正したらよいか，もうおわかりでしょう．Fig. 2.3が，その修正したものです．細かい点では多少の相違がありますが，〝COMMAND．COM〟の概念的な働きは大方このようになっています．

2.1.4　内部コマンドの動作

内部コマンドの動作は，ほかのコマンド処理の動作よりは比較的単純なものです．

たとえば，COPY コマンドは2つのファイルをオープン(または作成)して，一方のファイルからその内容を読み出して他方のファイルへ書き込むだけですし，DEL コマンドはシステムコールのファイル削除をコールするだけです．ほかのものも，たいてい数個のシステムコールを組み合せただけで実現できます．CHDIR，MKDIR，RMDIR に至っては，単にそのパラメータをシステムコールへ渡すだけで操作が完了してしまいます．このとき，パラメータの正当性のチェックはすべてシステムコール側で行われているので，〝COMMAND．COM〟側で行う処理は，レジスタに返されたエラーステータス番号に従って決まったエラーメッセージをコンソールに出力するだけです．もっとも，エラーハンドリングを簡単にするために，システムコールが強力になっているとも考えられます．

2.2　I／Oのリダイレクト

I／O のリダイレクトは MS-DOS の大きな特徴の1つですが，その実現法は標準入出力を利用しているために，いたって簡単に処理されています．

入出力の方向を切り換えるという概念が，MS-DOS ではどのように実現されているのか次に見ていきます．

2.2.1　リダイレクトの処理

I／O のリダイレクトの例として，次のようなコマンドを実行したとします．

```
SORT　<INFILE　>OUTFILE
```

コマンドの実行結果は，〝INFILE〟の内容をソートして〝OUTFILE〟に出力されますが，このとき〝COMMAND．COM〟の処理することは，実際には〝INFILE〟と〝OUTFILE〟をオープン(ファイルの入出力を可能にする)して，後はプログラムをロードして実行するだけです．

これだけで I／O のリダイレクトは完了し，ロード，実行されているプログラムでは，コマンド

に対する入出力が何から行われているかはまったく関与する必要はありません．つまり，〝COMMAND．COM〟側でリダイレクト用のファイルをオープンした後は，コマンドと入出力の方法は互いにまったく無関係なのです．

このようにI／Oのリダイレクトが簡単に実現できるのは，MS-DOSの(というよりは，そのもととなったUNIXの)標準入出力の概念設計がしっかりしているからだといえます．

2.2.2　標準入出力とリダイレクト

MS-DOSでは，一度オープンしたファイルはファイル・ハンドル(File Handle)という，BASICでいえばファイル番号にあたるもので扱われます．Table 2.1の0〜4は，それぞれの標準入出力に対応するファイル・ハンドルの値です．この値は，機械語レベルからでないとそのまま使用するということはありませんが，ここでは〝COMMAND．COM〟の説明のためにあえて引用します．

Table　2．1　標準入力と割当て番号

0	標準入力
1	標準出力
2	標準エラー出力
3	標準補助装置
4	標準プリンタ

たとえば，SORTコマンドが標準入力からデータを入力して標準出力へ出力するという動作は，ファイル・ハンドル0のファイルから入力してファイル・ハンドル1のファイルへ出力するということになります．つまりSORTコマンドは，現在の入出力が何から行われているかにはまったく関与せず，単に0のファイルから文字列を読んで1のファイルに書き込んでいるにすぎません．

いい換えれば，さきほどのSORTコマンドの例では，ファイル0とファイル1に対して，コンソールではなく〝INFILE〟と〝OUTFILE〟を割り当てておけばよいのです．BASIC·ふうに表現すると，標準状態で

```
OPEN "CON" FOR INPUT AS #0
OPEN "CON" FOR OUTPUT AS #1
```

となっているのを，標準入出力をリダイレクトするときには

```
OPEN "INFILE" FOR INPUT AS #0
OPEN "OUTFILE" FOR OUTPUT AS #1
```

としておけばよいのです．

こうしておいてから，システムコールのプログラムのロードと実行(ファンクション4BH)を使ってSORTコマンドを実行させます．このときロードする側(この場合COMMAND．COM側)でオープンされていたファイルや環境は，すべてSORTコマンド側へ引き継がれます．ですから，SORTコマンド側ではファイルのオープンの手続きをまったく行わずに，いきなりファイルの内容の読み書きがはじめられます．また，クローズの手続きも心要ありません．コマンド実行後，"COMMAND．COM" 側でクローズされます．

2.3 パイプ

本節では，リダイレクトの概念の延長であるパイプ機能の処理について見ていきます．

まず，パイプの基本動作を確認するためにFig. 2.4に示すようなコマンドを入力して，結果を見ることにします．

```
                    ……このDIRコマンドにより以下の出力が得られる
A>dir | dir

 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19
%PIPE1   $$$        0  84-04-19  11:03
%PIPE2   $$$      258  84-04-19  11:03   } "|" を用いた結果できたパイプ用の中間ファイル(テンポラリファイル)
       3 個のファイルがあります
 1172480 バイトが使用可能です
```

Fig. 2.4　DIR | DIR

このとき2番目のDIRは，1番目のDIRの実行結果を受け取りません．つまり，1番目のDIRの結果はまったく無視されます．このような使い方は，パイプを使うというより，単に2つのコマンドを連続実行させているにすぎません．

さて，Fig. 2.4の結果の出力では，"%PIPE1．$$$" とか "%PIPE2．$$$" なるファイルが存在しています．しかし，こんなファイルは作った覚えはありません．もう一度DIRコマンドでディレクトリを見てみます(Fig. 2.5)．

```
A>dir

 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19   } 中間ファイルはない！
       1 個のファイルがあります
 1173504 バイトが使用可能です
```

Fig. 2.5　中間ファイルの確認

確かにありません．そこでもう一度，さきほどと同じコマンドを入力してみます(Fig. 2.6)．

```
A>dir ¦ dir↵

 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
%PIPE1   $$$       258  84-04-19  11:03   中間ファイル．さきほどのものと比べると大きさが変わっているのがわかる
%PIPE2   $$$         0  84-04-19  11:03
       3 個のファイルがあります
 1172480 バイトが使用可能です
```

Fig. 2．6　DIR ¦ DIR

やはり，〝%PIPE？．$$$〟というファイルが2つあります．さきほどのものと比較してみると，ファイルのサイズが異なっています．というより入れ換わっているようです．

以上のことから，〝COMMAND．COM〟がどうやってパイプ処理を実現しているのかが推測できます．

まず，最初の〝DIR ¦ DIR〟を

```
DIR  >%PIPE2．$$$
DIR  <%PIPE2．$$$
```

と解釈して実行し，2番目の実行では

```
DIR  >%PIPE1．$$$
DIR  <%PIPE1．$$$
```

と解釈して実行しているのです．〝<〟や〝>〟の実現方法は前節で述べたとおりです．

どうして2つのファイルを交互に使うのかについては，次の例を考えればおわかりいただけると思います．

```
DIR  ¦  SORT  ¦  MORE
```

を実行するには，

```
DIR    >%PIPE2．$$$
SORT   <%PIPE2．$$$   >%PIPE1．$$$
MORE   <%PIPE1．$$$
```

と解釈します．つまり，パイプの処理では標準入力と標準出力の2つが同時に必要となるため，2つのファイルを使うのです．

MS-DOSでは，このファイルはカレント・ドライブのルート・ディレクトリに作られます．これをカレント・ディレクトリに作るようにするために，DEBUGを使ってちょっと細工をしてみましょう(Fig. 2.7)．"COMMAND．COM"もひとつのCOM型の実行ファイルにすぎないため，DEBUGで修正して実行することも可能です．

```
A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
       1 個のファイルがあります
 1173504 バイトが使用可能です

A>mkdir test↵
A>chdir test↵          ルート・ディレクトリ(¥)ではない別のディレクトリに移る

A>dir ¦ dir↵           ……パイプを使う

 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥test

.            <DIR>      84-04-19  12:00
..           <DIR>      84-04-19  12:00   パイプ用ファイル(中間ファイル)はない
       2 個のファイルがあります
 1171456 バイトが使用可能です

A>debug a:¥command.com↵   ……DEBUGで "COMMAND．COM" を一部変更する
-r↵                    ……サイズを調べる
AX=0000  BX=0000  CX=4326  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=1292  ES=1292  SS=1292  CS=1292  IP=0100   NV UP DI PL NZ NA PO NC
1292:0100 E9CD0B        JMP     0CD0
-s100 l4326 "PIPE"↵    ……パイプ用のファイル名(ASCIZ文字列)を捜す
1292:0B63              ……見つかった場所
1292:0B71              ……サイズ
-db50 130↵             ……表示させる
1292:0B50  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 5F  ................-
1292:0B60  3A 2F 25 50 49 50 45 31-2E 24 24 24 00 5F 3A 2F  :/%PIPE1.$$$._:/
1292:0B70  25 50 49 50 45 32 2E 24-24 24 00 5F 0B 6D 0B 00  %PIPE2.$$$._.m..
-eb61 "%"↵
-eb6f "%"↵             "/"(ルート用の印，この場合は "¥" と同じ)を "%" に変える
                       (ルートに作成するのをやめさせる)
-db50 130↵
1292:0B50  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 5F  ................
1292:0B60  3A 25 25 50 49 50 45 31-2E 24 24 24 00 5F 3A 25  :%%PIPE1.$$$._:%
1292:0B70  25 50 49 50 45 32 2E 24-24 24 00 5F 0B 6D 0B 00  %PIPE2.$$$._.m..
-g↵                    ……新しい "COMMAND．COM" を走らせる

Command ﾊﾞｰｼﾞｮﾝ 2.11

A>dir ¦ dir↵           ……パイプを使う

 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥test

.            <DIR>      84-04-19  12:00
..           <DIR>      84-04-19  12:00
%%PIPE1  $$$        0   84-04-19  12:01   カレント・ディレクトリに中間ファイルができた!!
%%PIPE2  $$$      302   84-04-19  12:01
       4 個のファイルがあります
 1171456 バイトが使用可能です

A>■
```

中間ファイルの名前(2つ分)

変わっている

Fig. 2.7　カレント・ディレクトリに中間ファイルを作る

なお，この新たに呼び出した〝COMMAND．COM〟からもとの〝COMMAND．COM〟へ戻るには，EXITコマンドを使います．

MS-DOSでは，このように中間ファイル(テンポラリファイル)を実際に作ってパイプ処理を実現しているので，名前の衝突や途中で中間ファイルが消されてしまうという事態が起こる可能性がありますが，これはシングルタスクのDOSでは仕方がないところでしょう．Fig. 2.8は，パイプ処理の過程で中間ファイルが破壊された場合のエラーメッセージです．ちなみにUNIXでは，〝|〟の両側に並べられたプロセス(個々のプログラムのこと)をマルチタスクで走らせ，それぞれをOSのカーネル(本体)内でパイプで結合しています．そのため，左側のコマンドがすべて終了しなくても右側のコマンドの実行に移ることができますし，パイプもたいていメモリ中で管理されているので，実行に要する時間はかなり速くなっています(もっともCPUやハードウェアの差に負うところも大きい)．

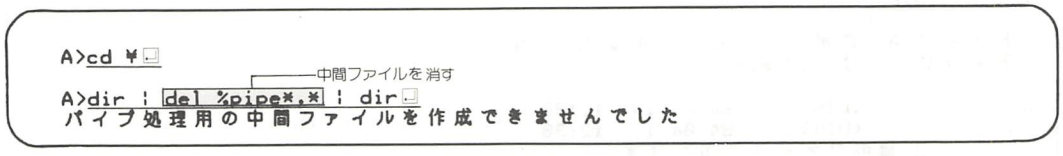


Fig. 2．8　パイプの壊し方

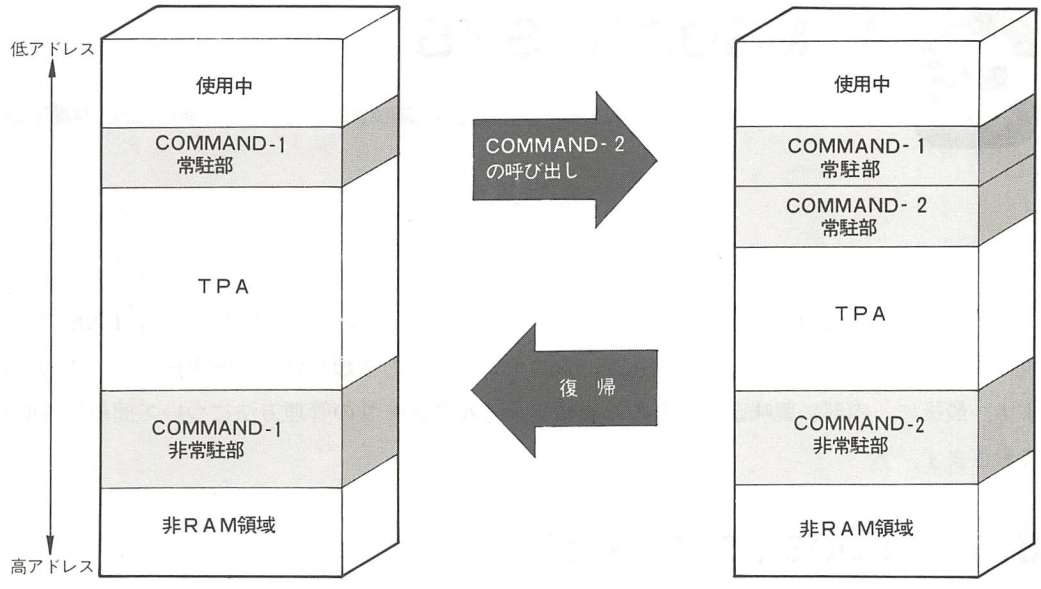
2.4　COMMAND．COMの再ロード

〝COMMAND．COM〟のメモリマップについては，すでに1.2で述べましたが，ここではもう少し詳しくみていきます．

DIRコマンドで〝COMMAND．COM〟の大きさを見ると，16Kバイト以上もあります．これだけの大きさのものがメインメモリに置かれると，128KバイトのRAMしか実装していない機種では，TPAがかなり小さくなってしまいます．そこでMS-DOSでは，〝COMMAND．COM〟を常駐部と非常駐部の2つに分けています．常駐部は約3Kバイトの大きさで，外部コマンドとしてロードされた〝COMMAND．COM〟がEXITコマンドを受け付けると，この〝COMMAND．COM〟が実行を終了し，〝COMMAND．COM〟を呼んだプログラムの方へ戻ります．このとき，当然常駐部も解放されてTPAとなります．

例として，〝COMMAND．COM〟から，新たに〝COMMAND．COM〟を呼んだときのメモリの状態をFig. 2.9に示します．

なお，〝COMMAND．COM〟は通常のCOMファイルと同じですので，実行は100H番地からはじまります．そして，非常駐部をメモリの最後部へロードするのは，この2番目の〝COMMAND．COM〟によって行われます．



Fig．2．9　COMMANDからCOMMANDを呼ぶ

〝COMMAND．COM〞を実行するときにPオプション(Permanent)を付けると，EXITコマンドが利かなくなります．つまり，メモリ中に本当に常駐してしまうわけです．システムによって最初にロードされる〝COMMAND．COM〞にはこのPオプションが付いているため，いつまでもメモリ中に存在しています．

〝COMMAND．COM〞が非常駐部を再ロードするか否かは，非常駐部が破壊されたか否かによりますから，強制的に再ロードさせるにはRAMの最後部付近にでたらめなデータを書き込みます(Fig．2．10)．

```
A>debug ……DEBUGを起動
-q ……ここで，ドライブのドアを開けてから〝q〞コマンドを実行．何も起らず，DOSへ戻る

A>debug ……ドライブのドアを閉めてからもう一度DEBUGを実行
-d0 140 ……PSPを見る
                                  ―― メモリの終了セグメント・アドレス＋1
11B3:0000  CD 20 00 40 00 9A EE FE-1D F0 4A 02 AB 0E 78 02   M .@..n~.pJ.+.x.
11B3:0010  AB 0E BA 04 D9 0D D9 0D-01 01 01 00 02 FF FF FF   +.:.Y.Y.........
11B3:0020  FF FF FF FF FF FF FF FF-FF FF FF FF A7 0E 05 2C   ............'..,
11B3:0030  AB 0E 00 00 00 00 00 00-00 00 00 00 00 00 00 00   +...............
-f3000:0 10 0 ……メモリの最後の64Kバイトに0をかく
-q ……ドライブのドアを開けてから〝q〞コマンドを実行すると…

ドライブの準備ができていません くドライブ A:>
正しいディスクをカレントドライブに差し込み，
どれかキーを押してください

A>■
```

〝COMMAND．COM〞を再ロードしようとしたが，ドライブのドアが開いていたため，こうなってしまった

Fig．2．10　COMMAND．COMの再ロード

3章 MSDOS.SYS

まず最初に〝MSDOS.SYS〟の機能の概要を述べた後，環境やCOMファイル，EXEファイルについて解説します．次に内部割込み，システムコールの種類や働き，使用法について解説します．最後に，内部に興味がある方のためにファイルやメモリの管理方法について簡単に説明しておきます．

3.1 MSDOS.SYSの概要

〝MSDOS.SYS〟は，MS-DOSの中核としてMS-DOSの性格のほとんどを形成している部分であり，主な機能として以下のものが挙げられます．

(1)ファイル管理
(2)メモリ，プロセス管理
(3)周辺装置の制御

MS-DOSは，シングルユーザー／シングルタスクのDOSであるため，〝MSDOS.SYS〟の主な機能は(1)のファイルを管理することにあります．ファイルの管理とは，ファイルのオープン，クローズ，読出し，書込み，作成，削除，名前変更，属性のセットほか，ファイルの取扱いに関するすべてのことをいいます．また，ディレクトリの作成，削除なども，ファイルの取扱いに関するものといってもよいでしょう．

コンソールに対する入出力も，MS-DOS ver. 2.0ではファイルの入出力の一種となっています．MS-DOS ver. 1.25でも，コンソール(CON)や他のデバイス名もディスク上のファイルと同じようにオープン，クローズできましたが，ver. 2.0ではこの考え方(デバイスとファイルを同一視する)がさらに徹底したものになっています．

しかし，MS-DOS ver. 2.0のメモリやプロセスの管理は，現状では十分に活かされているとはいえません．ver. 2.0はシングルタスクのDOSなので，これらの機能が本当に役に立つのは将来のバージョンにおいてでしょう．

周辺装置の管理機能としては，ディスクとの直接入力や時間制御用クロックの管理などがあります．

〝MSDOS．SYS〞のこれらの機能はシステム側のサービスルーチンとしてユーザーに開放されており，MS-DOS上で機械語プログラムを開発する際には，ユーザープログラムとMS-DOSのインターフェイスを行います．

3.2　内部割込みの機能

8086CPUでは，ユーザー定義ソフトウェア割込みとして，20Hから FFHまで使用できますが，MS-DOSではこのうち20Hから3FHの32個を使用しています．このソフトウェア割込みを介して〝MSDOS．SYS〞を呼び出し，その機能を使うことを，以降，内部割込みと呼びます(Fig. 3.1)．

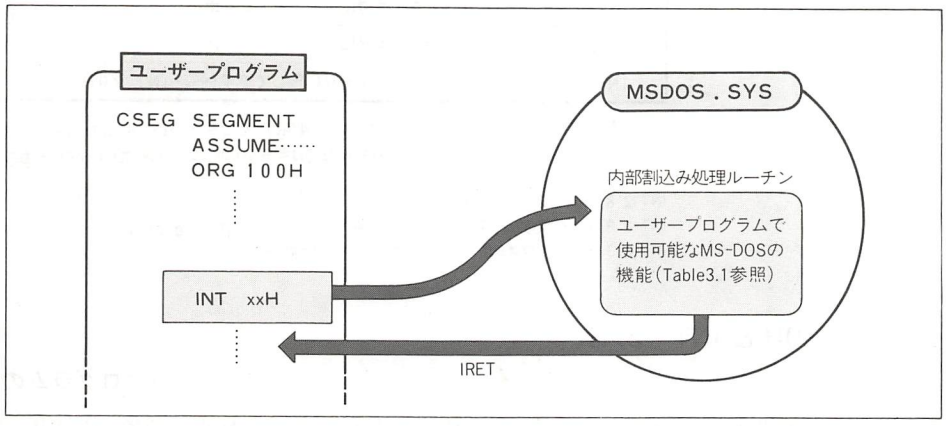


Fig. 3．1　内部割込み

ソフトウェア割込みというのは一種のジャンプ命令ですから，FAR CALL※命令で代用できますし，こちらの命令の方がクロック数が少ない(FAR CALLが28クロックに対してINTは50クロック以上)のですが，INT命令を使うメリットはそれを補って余りあるものがあります．それは，INT命令を使うことにより得られるポータビリティです．つまり，多機種への汎用性を備えたソフトウェアとなり得るのです．JMP命令やCALL命令の場合，その命令中にアドレスを持っているため，ジャンプ先が変わるたびに命令自身を書き換えなければなりませんが，INT命令を使用していれば，DOS自身が割込みベクトルテーブルをセットするだけでプログラム自身に修正を加える必要はありません．また，ユーザープログラムとそれを管理するDOSを分離するという点からも，このINT命令の使用が望まれます．

※本書ではセグメント間CALL命令をFAR CALLとしています．

MS-DOSにおける各内部割込みの機能をTable 3.1に示しておきます．

MS-DOSのソフトウェア割込みのうち，内部割込みとしてユーザーが利用できるものはINT 20H～INT　27Hですが，このうちINT　22H～INT　24Hは，MS-DOSがダブルワードのアドレス(セグメント＋オフセット)を置くためだけに使用しているため，命令として使うことはできません．

Table　3．1　内部割込み一覧

割込み番号	機能
INT　20H	プログラムの終了(非常駐終了)
INT　21H	システムコール(ファンクション・リクエスト)
INT　22H	プログラム終了アドレス
INT　23H	CTRL-Cアドレス
INT　24H	致命的エラー処理ルーチンのアドレス
INT　25H	物理セクタ番号によるディスク読出し
INT　26H	物理セクタ番号によるディスク書込み
INT　27H	プログラムをメモリに残して終了(←→INT　20H)

(注)INT　28H～INT　3FHは，MS-DOSの内部使用のために予約されているので，ユーザープログラム中で使用してはならない．それらの中で，仕様は明らかではないが用途の決まっているものを次に示しておく

INT　28H……プリンタ・スプーラ(PRINT．COM)用
INT　29H……文字出力用(MS-DOSの将来のバージョンでは使用できなくなる)
INT　33H……マウス(ポインティング・デバイス)制御用

3.2.1　INT　20H & INT　27H

(プログラムの終了)

この2つの内部割込みは，ともにプログラムの終了のために用いられるもので，BASICでいえば，STOPやENDに相当します．プログラム中でこれらを用いることにより，制御は，このプログラムを呼び出したもとのプログラムに戻ります．両者の機能の違いはFig.3.2に示すように，INT　27Hがプログラムをメモリに常駐させておくために用いるのに対し，INT　20Hはそうではない，つまり通常のコマンドのために使われます．

INT　27Hを使用すると，それ以降プログラムがメモリに常駐したままになり，当然その分だけ使用できるメモリ領域が減ります．ですから，次にコマンド(プログラム)のロードされる位置が後ろの方へずれることになります．

INT　20Hでプログラムが終了する場合はそのプログラムで使用していた領域が解放されることになり，次にロードされるプログラムは，通常今のプログラムが使用していた領域と同じ位置からロードされます．

INT　27Hを利用すると，メモリに常駐したままプログラムが終了するので，この特徴を利用するとユーザーが新たにデバイス・ドライバを作成することが可能となります．もっともver. 2.0では，"CONFIG．SYS" ファイルに登録する方法でデバイス・ドライバを実現しているために，

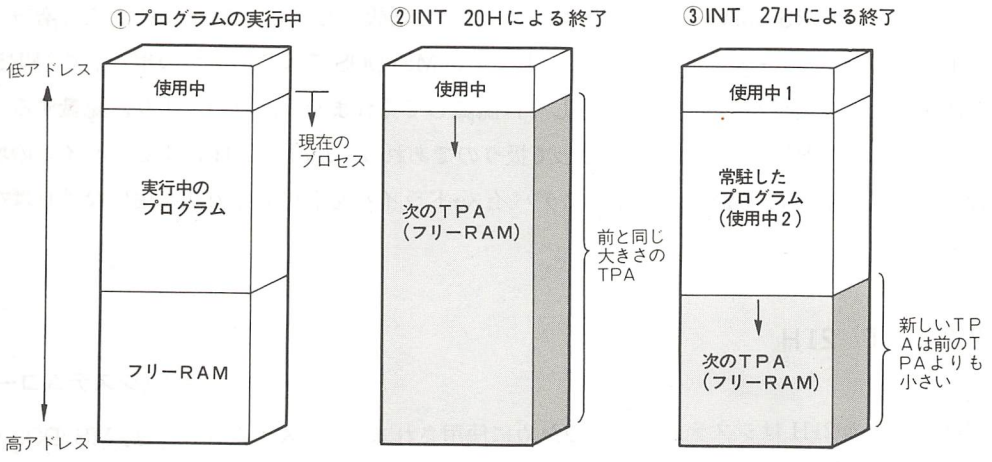


Fig. 3．2　INT 20HとINT 27H

実際にINT 27Hを使用することはないでしょうが，MS-DOS ver. 1.25との互換性を保つために残されていると考えられます．

たとえばシステムにマウスを付加した場合，マウスの動きを読み取ったり，あるいはその動きに応じてマウスカーソルを画面上に表示，移動させなければなりません．そのためのプログラムは，入出力制御部(IO．SYS)の一部として作らなければなりませんが，最初からそのようなものを組み込んでおくと，マウスのないシステムではメモリやディスクをむだに使うことになります．また"IO．SYS"は，通常MS-DOSを供給している各メーカーで作られるので，ユーザーが新たに独自のデバイスを付加することが不可能になります．

このような場合にINT　27Hの機能を使えば，ユーザーがいつでも自由にデバイス・ドライバやそのアプリケーション・プログラムが使用できるようになります．

ところで，ここで注意していただきたいのは，このようにして作ったデバイス・ドライバの使用方法です．このデバイス・ドライバがメモリにロードされるとき，つねに一定のメモリ・アドレスにロードされるということはありませんし，ロード位置を指定することもできません．そのため，このデバイス・ドライバ内のルーチンをFAR CALL命令で呼ぶということはできないので，デバイス・ドライバ内のルーチンを呼ぶためには，ソフトウェア割込みINT　xxHを利用するのが一般的です．ただし，8086／8088CPUやMS-DOSで予約されているソフトウェア割込みは使用できません．またシステムによっては，このほかにも使用できない(システムで使用済みの)ソフトウェア割込みがあるため注意する必要があります．なお，このINT命令用のベクトルは，デバイス・ドライバ自身でINT　27Hで終了する前にセットしておかなければならないのはいうまでもありません．

もう1つ注意していただきたいのは，このようにして増設したデバイス・ドライバはMS-DOSで扱っているデバイスとは何の関連もないということです．たとえば，ユーザーがこの方法でプ

リンタ用ポートとRS-232C用ポートをユーザー自身で増設したとします．しかし，この増設したポートとそのデバイス・ドライバがあったとしても，MS-DOSではこの2つのポートを〝PRN2〟や〝AUX2〟といったデバイスファイルとしては認識してくれません．というより，認識することができません．〝PRN2〟や〝AUX2〟として扱うのであれば，INT　27Hによるデバイスの増設ではなく，〝CONFIG．SYS〟ファイルにデバイス・ドライバを登録する方法を用いなければなりません．

3.2.2　INT　21H

（システムコール）

内部割込みの21Hはシステムコールのために使用されます．システムコールは，MS-DOS ver. 2.0では70個以上も定義されており，AHレジスタにそのファンクション番号を入れておいてからINT　21Hを使います．また，必要に応じてほかのレジスタにもパラメータを設定しておかなければなりません．

システムコールの主な機能としては，コンソールI／O，ファイルの扱い，メモリの管理などがあります．

なお，個々のシステムコールの機能については，7章で詳しく解説します．

3.2.3　INT　22H，INT　23H & INT　24H

（終了アドレス，CTRL-Cアドレス，致命的エラーによる中断アドレス）

これらの3つのINT命令用のベクトルは，プログラムの終了時や中断時のジャンプ先アドレスを格納するために用いられます．これらはアドレスを保持しているだけで，INT命令用ベクトルではありません．ですから，プログラムを終了させるためにINT　22H命令を用いることはできません．

これら3つのベクトルは，通常〝COMMAND．COM〟内のそれぞれのルーチンのベクトルがはいっていますから，多くのプログラムは実行が終了すると〝COMMAND．COM〟へ戻り，コンソール入出力を実行中にCTRL-Cを押すと，やはり〝COMMAND．COM〟へ戻ります（BREAKフラグがONになっていると，コンソール入出力だけでなくMS-DOSの他のシステムコールを行った場合にもCTRL-Cアドレスへ制御が移る）．

INT　24Hの処理ルーチンは，通常〝COMMAND．COM〟内に用意されており，ディスクエラーなどが起こったときに，Abort（中断）するかRetry（再試行）するかIgnore（無視して続行，つまり処理が正常に終わったものとみなす．たとえばディスクからの読出しならば読んだつもり，書込みならば書いたつもりなど）するかをユーザーに問い合せるルーチンとなっています．

INT　23HのCTRL-Cによる中断は，つねにキーボードのチェックをする必要がないため，プログラムの中ではいくつかの使い方が考えられます．たとえば対話型のプログラムでは，ネスト

したコマンド(あるコマンド列の中からAを選び，その中のコマンドとしてBを選び，さらにその中のコマンドからCを選び……というような階層構造のコマンド)の中から一気に外へ抜けてトップレベル(いちばん外側のコマンド)へ戻るというアプリケーションに威力を発揮するでしょう．

INT　24Hは，エラーが起こった場合のリカバリ(復旧)用に役に立つでしょうが，一般にこのベクトル(24H)だけはもとのまま使用する方がよさそうです．というのは，このルーチンではMS-DOSのシステムコール処理時に起こったエラーしか処理できず(このルーチンはMSDOS．SYSからしか呼ばれない)，処理の内容もAbort(中断)，Retry(再試行)，Ignore(無視)の3つのうちからしか選べません．ですから，プログラムごとにこのルーチンを書くメリットはあまりありません．もっとも，Multiplan(マイクロソフト社製の作表言語)のように高度に対話的で，よけいなメッセージが予期せぬときに出ては困るようなプログラムはその限りではありません．ちなみに，Multiplanでは，処理のRetryかAbortのみができるようになっています．これは，無意味なデータで処理を続行しても(文字どおり)意味がないからでしょう．

3.2.4　INT　25H & INT　26H

(アブソリュート・ディスクリード，アブソリュート・ディスクライト)

この2つの内部割込みにより，ディスク上のファイルには無関係に任意のセクタの読み書きができます．DISKCOPYコマンドなどのユーティリティでは，この2つの内部割込みが使われています．

システムコールのファンクション2EHを使ってベリファイフラグをONにしておくと，INT 26Hを使った場合は自動的にデータのリードアフターライトを行ってディスクに書き込まれたデータが正しいかどうかを調べてくれます．ただし，このときディスクの書込み速度が通常の半分以下になるので，通常はベリファイフラグはOFFにしておいてもかまわないでしょう．

3.3　MS-DOSにおけるプログラムの実行(メモリモデル,PSP,環境)

MS-DOSの管理下でプログラムが実行される場合，DOSとの通信用データ領域(PSP)やプログラムの実行環境が重要な働きをしています．また，プログラムはそのファイルの型によってメモリ上での展開のされ方が異なり，アセンブラでプログラムを記述する場合には，それらの点を考慮しなければなりません．

3.3.1　MS-DOSにおけるプログラムの構造

8086CPUのメモリ空間は1Mバイトありますが，4つのセグメント・レジスタでアクセス可能な領域は最大256Kバイトになります．256Kバイト以上のメモリを使用したい場合は，プログラ

ム実行時に何らかの方法でセグメント・レジスタの値を変えなければなりません．また，8086CPUではセグメント・レジスタへのロードやセーブは特権命令ではありませんので，セグメント・レジスタの変更はDOSが行うよりもユーザーのプログラム自身で行った方が，速度や所要メモリ容量の点で効果的です．

しかし，あまり頻繁にセグメント・レジスタを更新すると(たとえば，数命令ごとや違う変数ごとやデータ領域のアクセスごとに)，プログラムの実行速度がたいへん遅くなりますし，プログラムの見通しも悪くなります．

そこで，ある程度まとまりを持ったプログラムやデータを1つにまとめ，セグメント・レジスタを変更しなくてもアクセスできるようにします．当然これらの大きさは64Kバイトまでに制限されますが，ほとんどのプログラムはこれで用が足ります．仮にプログラムやデータが64Kバイトを超えるようであれば，プログラム自身でセグメント・レジスタを更新することによってアクセスすることができます．

一見したところ今述べたような方法でも問題がないように思われますが，これには実は重大な欠点があります．というのは，プログラムを作成する際，つねに64Kバイトを超えたか否かを意識していなければならないからです．

そこでMS-DOSではプログラミングに際して，物理的なまとまりではなく，論理的なまとまりを意識してプログラムを作成することができるように，プログラムの構成単位とその依存関係が考えられています(Fig.3.3)．

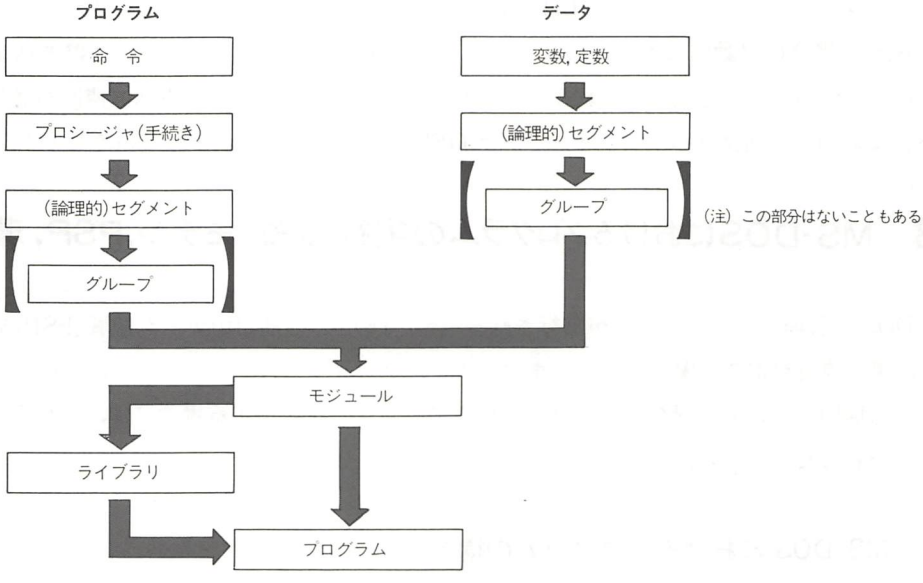


Fig. 3．3　プログラムの階層構造

プロシージャとは，いくつかの命令から成る１つのサブルーチンのことであり，通常 CALL 命令で呼ばれ，RET 命令によって実行を終了します．

セグメントは，いくつかのプロシージャやデータ領域から成るプログラムの論理的集合です．実行時には，8086CPU の持っているセグメント・レジスタにそのセグメントのベース・アドレスが入れられますが(ユーザープログラムの中で設定する)，使用できるセグメントの数に制限はありません．どのセグメントをどのレジスタに対応させるかは，通常ユーザーが自由に決められます．ですからこのセグメントというのは，ハードウェアのセグメントではなく，プロシージャやデータの論理的集合であると理解してください．

セグメントは，いくつか集めて１つの大きなセグメントとして扱われますが，これをグループと呼びます．また，セグメントやグループを集めたものをモジュールと呼び，これがアセンブルの単位となります．

このモジュールを集めるとプログラムとなります．プログラム・ライブラリはモジュールの集まりですが，そのままでは実行できないという点でプログラムとは異なります．そして，リンカによって，必要なモジュールのみがプログラムとリンク(結合)されます．

3.3.2　メモリモデル

MS-DOS の管理下で動くプログラムには以上で述べたような階層構造が要求されますが，こうしたプログラムは，そのリロケーション情報の有無によってメモリ上での展開され方が異なっています．

○ EXE モデル

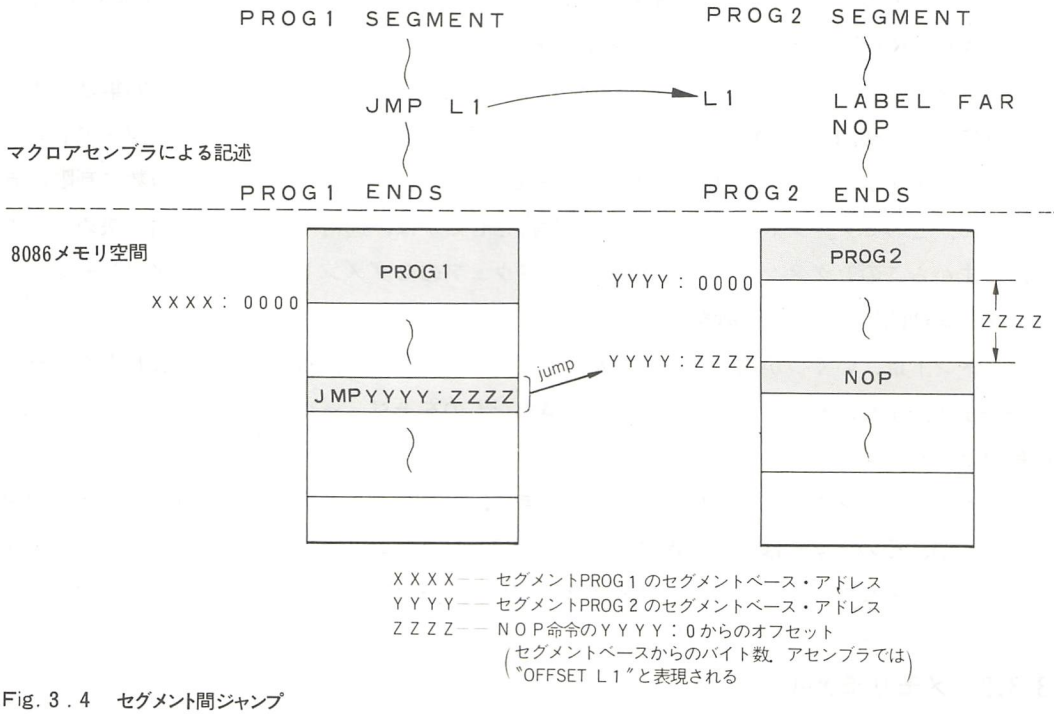
前項で述べたような構造を持ったプログラムがメモリへロードされる場合，セグメントを単位としてロードされ，各セグメントは 8086CPU のセグメント境界に合せてロードされます．ロードされるセグメントの位置は，プログラムのアセンブル時やリンク時にはどこのメモリ領域が使用可能であるかを知るすべがないためにロード時にならないと決定できません．

このような場合，セグメント間参照をどうするかが問題となります．今，次の Fig．3．4 のように，セグメント PROG1 から，セグメント PROG2 内のラベル L1 へジャンプするとします．

このとき，L1 のオフセット ZZZZ の値はアセンブル時に決まります．なぜなら，ロードはセグメントを単位として行われるので，セグメント内の構成はロードしても変わらないからです．

一方，ジャンプ先のセグメント・アドレス YYYY は実行時(ロード時)でないとわかりません．アセンブル時やリンク時にはどこへロードされるかわからないからです．

このため，正しく JMP 命令を実行させるためには，実行時(前)に YYYY の値を定め，JMP 命令の中へ組み込んでやらなければなりません．このような操作のことを〝リロケーション(再配置)〟と呼び，プログラムがメモリへロードされるたびに行う必要があります．



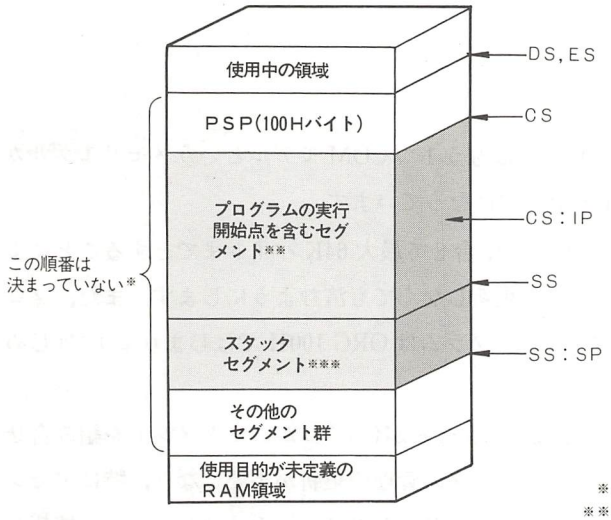
Fig．3．4　セグメント間ジャンプ

リロケーションは何もJMP命令だけでなく，ダブルワードのポインタやセグメントベースのロードなどをしている命令，データすべてについて必要となります．ただし，プログラムがセグメント間に何の関連も持っていない場合(セグメントベースの参照すらない)，このリロケーションは必要ありません．

MS-DOSでは，このような構造を持ったプログラムを〝EXEモデル〞と呼び，〝．EXE〞の拡張子を持ったファイルに入れられています(EXEファイル)．EXEモデルのプログラムの，メモリでの様子を表したのがFig．3．5ですが，CS，DS，ES，SSの各レジスタは，それぞれ8086CPUの4つのセグメント・レジスタの初期値を示します．つまり，ES，DSレジスタはPSPを指しており，CSレジスタはいちばん最初に実行される命令のセグメントを，また，SSレジスタはスタック属性を持ったセグメントを指しています．

EXEモデルのプログラムの場合，セグメント・レジスタの更新はプログラム自身で行わなければなりません．たとえば，あるセグメントが次のように宣言されているとします．

```
DATA    SEGMENT
VAR1    DW      1
ARRY1   DB      10  DUP  (？)
DATA    ENDS
```



※ロードされるセグメントの順番は，各セグメントに付けられたクラス名（マクロアセンブラの章を参照）と，LINKコマンドにより決定される
※※実行開始点は，マクロアセンブラでは，END文で指定する
※※※スタックセグメントにはSTACKという属性が付いていて，その他のセグメントと区別される．なお，ＳＰはつねにSTACKセグメントの最後に合せられる

Fig. 3．5　EXEモデルのメモリマップ

このセグメント中(DATA という名のセグメント)の VAR1 や ARRY1 をアクセスするためには，4つのセグメント・レジスタのどれかへセグメントのベース・アドレスを設定しなければなりません．DS レジスタにセットする場合は次のようになります．

```
MOV     AX, DATA   ;DATA は DATA セグメントのセグメントベース・アドレスを示す
MOV     DS, AX
```

これで，DATA セグメント中で定義したものがアクセスできるようになります．

EXE モデルは 8086CPU のアーキテクチャを最も忠実に反映したメモリモデルであり，プログラムの大きさに影響を受けることなく記述することができます．プログラムのコードやデータ部分がプログラムの論理的な構造に従ってセグメントやモジュールに分割されていれば，64K バイト単位のハードウェアの制約はあまりありません．もっとも，1つで 64K バイトを超えるような大きさのデータタイプ(配列など)に対しては，効率的なアクセスは行えません．

このように，EXE モデルはたいへん強力なメモリモデルですが，実は大きな欠点があります．それは，あまりにも構造が複雑であるという点です．つまり何をするにもまずセグメント・レジスタの更新を必要としますし，今どのセグメント・レジスタにどのセグメント・アドレスがはいっているかをつねに意識しておかないとアセンブラが誤ったコードを出力する可能性があります．また，つねにスタック・セグメントを定義しなければなりませんし，PSP へアクセスするにもそのつどセグメント・レジスタの更新が必要となるでしょう．また，もう1つの欠点として，

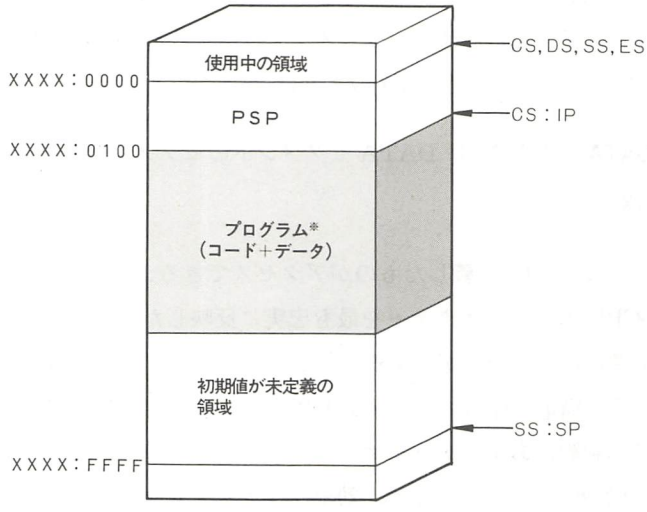
プログラムのロードの遅さ(リロケーションが必要なため)やプログラム格納用ファイルが大きくなるということが挙げられます．

○ COM モデル

EXE モデルの欠点を補うために，MS-DOS ではもう１つ COM モデルというメモリモデルが用意されています．これは，Fig．3．6 のような構造になっています．

つまりプログラム(コード＋データ＋スタック)は，合せて最大 64K バイトまでとすることにより，プログラム実行中にセグメント・レジスタを更新しなくても済むようにします．また，プログラムの最初に 256 バイトの PSP があるため，プログラムは ORG 100H ではじまります(はじめなければならない)．

COM モデルでは，セグメントは１つしか定義できませんが(いくつかのセグメントを組み合せて１つにしてもよい)，その代わりセグメント・レジスタの管理や更新が不要になり，特にアセンブラを使ってプログラムを書くのが容易になります．また，セグメントのリロケーション情報が不要なため，COM ファイルには本来のプログラム部分しかはいらず(リロケーション情報は含まない)，ファイルが小さくなります．



Fig．3．6　COMモデル

なお，COMモデルのすべてのセグメント・レジスタは，実行開始時には同一のアドレスを指しており，IPは100Hに，SPはFFFEHを指しており，スタックのいちばん上のワード(FFFEH, FFFFH)には0000Hがはいっています．これは，RET命令を実行するとIPが0になるようにするためです．これによる効果についてはPSPのところで触れます．

MS-DOSには，以上で述べたEXEとCOMの2つのメモリモデルがありますが，これらは通常，アセンブラ・プログラムはCOMモデルに，コンパイラによるプログラムはEXEモデルにというように使い分けられます．アセンブラしか使わないプログラムは，EXEモデルの構造はかなり複雑なので，あまり利用されません．また，アセンブラを使って64Kバイト以上ものコードを書くのは異常です．データが64Kバイト以上になるのはEXEモデルでもどうしようもないため，EXEとCOMのどちらを使ってもかまいません．別にCOMモデルだからといってセグメント・レジスタを書き換えてはいけないということもありません．これらはあくまでも原則なので，必要ならば自分の好きなように変更してもよいことになっています．

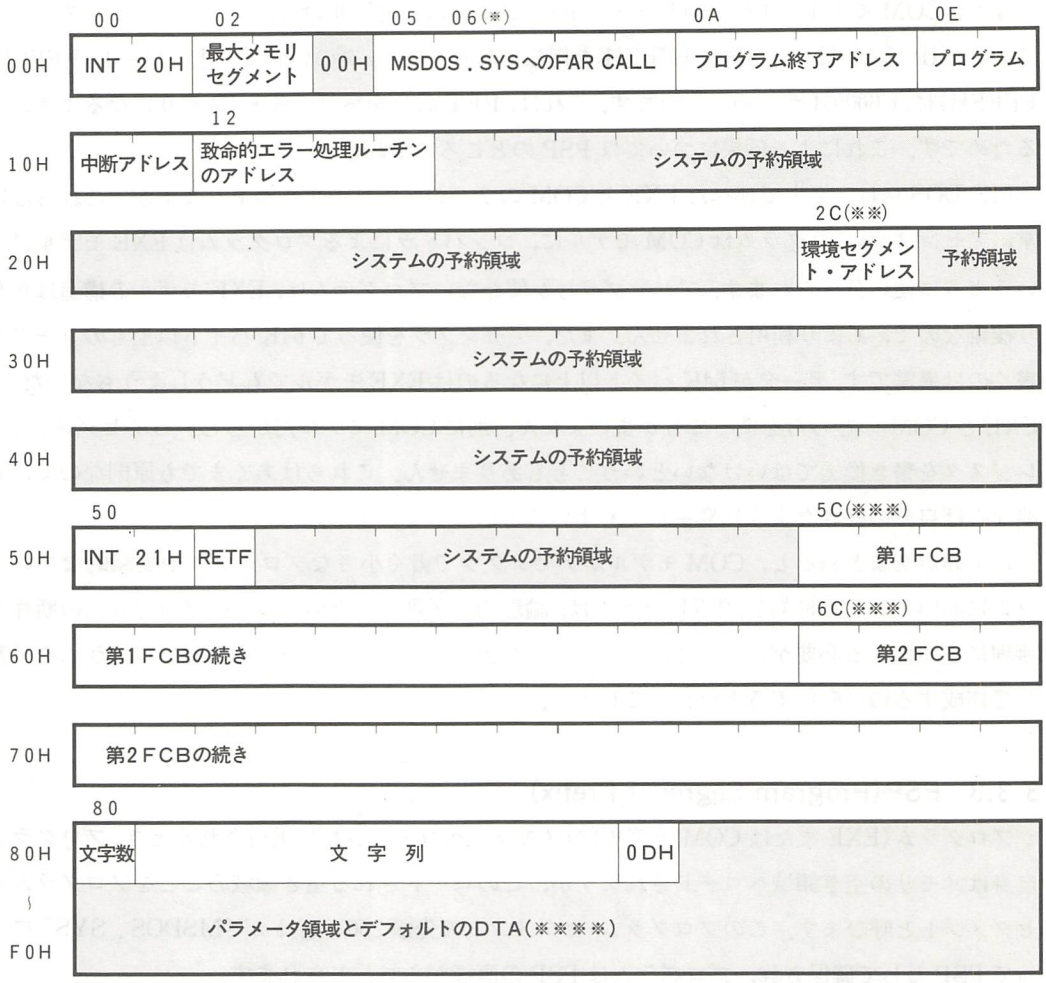
以上の点をまとめると，COMモデルがアセンブラで書く小さなプログラムや実験的なプログラムに向いているのに対し，EXEモデルは，論理的に区別されたいくつかのプログラムの断片を無理に1つにする必要がないため，コンパイラが出すオブジェクトや多人数でプログラムを分割して作成するのに向いてるといえるでしょう．

3.3.3　PSP(Program Segment Prefix)

プログラム(EXEまたはCOMモデル)がメモリ上へロードされて実行されるとき，プログラム自身はメモリの空き領域へロードされますが，このロードされる空き領域のことをプログラム・セグメントと呼びます．このプログラム・セグメントの先頭256バイトが〝MSDOS．SYS〟によってPSPとして確保され，プログラムはPSPの直後からロードされます．

PSPは，プログラムとシステム(MSDOS．SYS)の橋渡しの領域であり，〝COMMAND．COM〟から渡されたパラメータの文字列を格納しておくのが主目的です．PSPには，このほかに，後述する終了や中断のアドレス，FCB, DTA, 〝MSDOS．SYS〟の作業領域などが含まれています．

PSPの構成は，Fig. 3.7が示すようになっており，そのアドレスは8086CPUの20ビットのアドレスで表すとｘｘｘｘ0Hから100Hバイトとなっています．つまり〝セグメント＋オフセット〟の形式では〝ｘｘｘｘ：0000〟となり，つねにセグメントの境界から置かれているのです．ですから，PSPの内容をアクセスするためには通常DSレジスタかESレジスタにPSPのセグメント・アドレス(xxxx0H)を入れておき，そこからのオフセットでアドレスを指定します．



※オフセット06Hからの1ワードにはセグメント内で使用可能なバイト数がはいっている（COMのときに意味がある）
※※環境を表す文字列のはいっている領域の開始セグメント・アドレス．オフセットは0からはじまる
※※※2つのFCBは10Hバイトしかはなれていないため，どちらか一方のみが有効
※※※※DTAとして使うとパラメータの文字列は破壊される

Fig. 3．7　PSPの構成

○ PSP の内容

00H～01H

オフセット 00H からの 2 バイトには，INT　20H のコード(CDH，20H)がはいっています．これは，プログラムを終了させるときに用いられます(詳細は後述)．

02H～03H

オフセット 02H からの 1 ワード(8086CPU では，1 ワードは下位バイト，上位バイトの順にはいっている)には，使用できるメモリの最上位の〝セグメント・アドレス＋1〟がはいっています．

たとえば, RAM が 00000H～7FFFFH の 512K バイト実装されているとします. このとき, オフセット 02H には 8000H がはいります(Fig. 3.8).

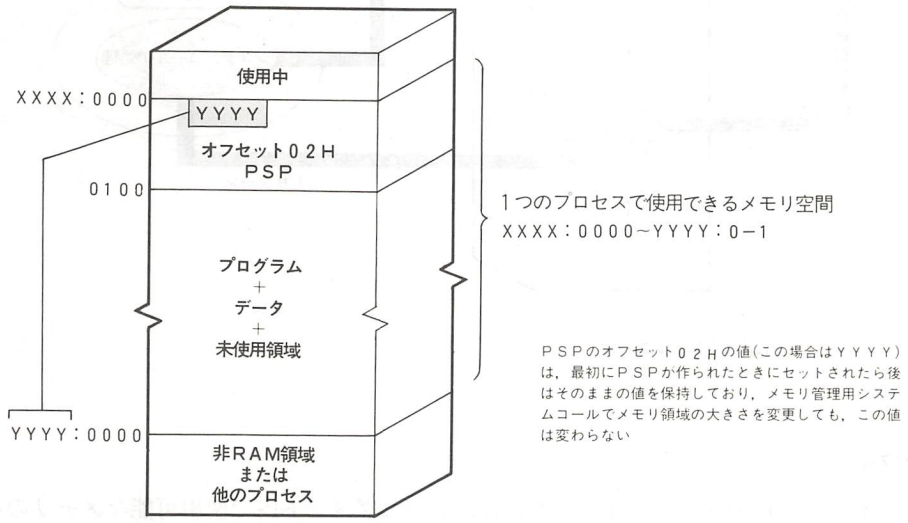


Fig. 3．8　PSPと使用可能なバイト数

05H～09H

PSP のオフセット 05H からの 5 バイトには, "MSDOS．SYS" への FAR CALL 命令がはいっています. MS-DOS では, システムコールは基本的にはファンクション番号を AH レジスタに入れて INT　21H を行いますが, いくつかのシステムコールでは, ファンクション番号を CL レジスタに入れてこの PSP のオフセット 05H をコールすることでも同じ効果が得られます. このシステムコールの方法は CP／M-80 との互換性を保つためにありますが, この方法が使えるのは AL レジスタにパラメータを渡さない 00H～24H のファンクションに限られます. また, このコール方法を用いると AX レジスタはつねに破壊されます.

オフセット 05H には FAR CALL 命令がはいっているのに, オフセット 0AH には RET 命令がありません. これは, 制御が直接"MSDOS．SYS"からユーザープログラム内へ戻ることを意味しています. そのため, 05H へはセグメント内の CALL(NEAR CALL)命令を使わなければなりません. FAR CALL 命令ではスタックを調整しなくてはならないので, EXE モデルでは使用しない方がよいでしょう. もっともこの FAR CALL 命令は CP／M-80 との互換性を保つためにあるのですから, COM モデル以外で使うのは邪道でしょう. この機能を使った場合のプログラムの流れを Fig. 3.9 に示します.

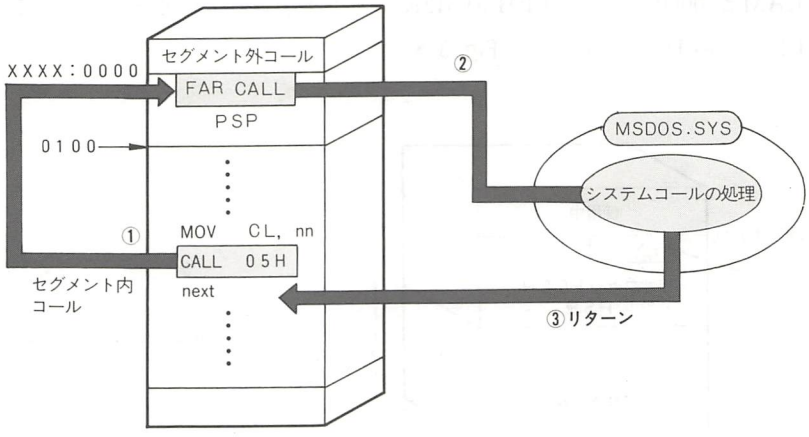


Fig. 3．9　CALL　5によるシステムコール

06H～07H

PSPのオフセット06Hからの1ワードには，そのセグメント内で使用可能なメモリのバイト数がはいっています．つまり，PSPを含むセグメント内でユーザーが使用可能なバイト数のことを指します．

なお，さきほどオフセット05Hから5バイトのFAR CALL命令があると述べましたが，このオフセット06Hの利用可能最大メモリバイト数はその命令の一部と重なっています．つまりこの値は，FAR CALL命令のセグメント・オフセット部となっているのです．

0AH～15H

PSPのオフセット0AHからは，ダブルワードのポインタ(セグメント・アドレス＋オフセット)が3つ連続してはいっています．これらは，実はINT　22H～INT　24Hの割込みベクトルのコピーです．PSPが作られるときこれらが8086CPUの割込みベクトル領域からコピーされ，プログラムの実行が終わった時点でもとの領域へ戻されます．つまり，実行中は割込みベクトルをいくら変更してもかまわないのです．PSP中に保存されている3つのベクトルがプログラム終了時にもとに戻されるので，プログラムごとに独自のCTRL-Cの処理ルーチンを持つことができます．また，プログラム終了用ベクトルを操作することにより，プログラム中から別のプログラムを起動することができます(詳細は後述)．

2CH～2DH

PSPのオフセット2CHからは，環境のセグメント・アドレスがはいっています．環境とは，コマンドのサーチパスや"COMMAND．COM"のプロンプトマークなどの設定状況のことを指します．

50H～52H

オフセット 50H からは，次の 3 バイトの命令列がはいっています．

```
INT　21H
RETF
```

INT　21H はシステムコール用の内部割込みですから，ユーザーは，システムコールを行うために PSP のオフセット 50H へ FAR CALL してもよいということになります．現在の MS-DOS では，ここには INT　21H の命令コードがはいっているだけなので，単にプログラムで INT 21H を使用するのと大差ありません．しかし，もし遠い将来(？)システムコールの方法が変わってしまうようなことがあっても，プログラム中で FAR CALL 命令でこの PSP＋50H を呼ぶようにしていれば，そのプログラムはあまり変更する必要がありません．PSP のオフセット 00H からオフセット 5BH まではユーザーが書き換えてはいけないことになっているため(読むだけならよい)，MS-DOS の将来のバージョンではこの 50H からの内容が変わる可能性は十分にあります．マニュアル中では，この 50H を用いたシステムコールについては以下のようにしか書いてありません．日本語に訳すと原文の微妙なニュアンスが薄れるのでそのまま掲載しておきます．

> Since this is a call and not an interrupt, MS-DOS may place any code appropriate to making a system call at this position(PSP＋50H). This makes the process of calling the system portable.

5CH, 6CH

PSP のオフセット 5CH と 6CH には FCB(ファイル・コントロールブロック)があり，起動されたプログラムに対する第 1 と第 2 のパラメータがセットされています．FCB についてはのちほど詳しく述べますが，簡単にいうと，ファイルに対してオープン，クローズ，読出し，書込み，削除などの操作をするときに用いられる一種の作業領域です．FCB の大きさは，普通に使われるものは 37 バイトあり，その内容は，11 バイトのファイル名以外はほとんどシステム(MSDOS．SYS)側の作業領域の集まりです．蛇足ですが 5CH, 6CH などという切れの悪い(？)アドレスにあるのは，CP／M 以来の伝統です．

ところで，FCB の大きさは 37 バイトあると述べましたが，5CH と 6CH では 16 バイトしか差がありません．ファイル名だけなら 16 バイトでも間に合いますが，本当に FCB として使うためにはどちらか一方の FCB のみが有効となります．

80H

PSP のオフセット 80H からは，コマンドに渡されたパラメータの文字列がはいっています．正確にいえば，80H にはパラメータの文字数がはいっています．たとえば，0 はパラメータがない

ことを示し，1ならば1文字あることを示すという具合いです．また81Hからは実際の文字列がはいり，最後には0DHがはいっていますが，この0DHは文字数には含まれません．

PSPの実際のダンプ例をFig. 3.10に示しておきます．

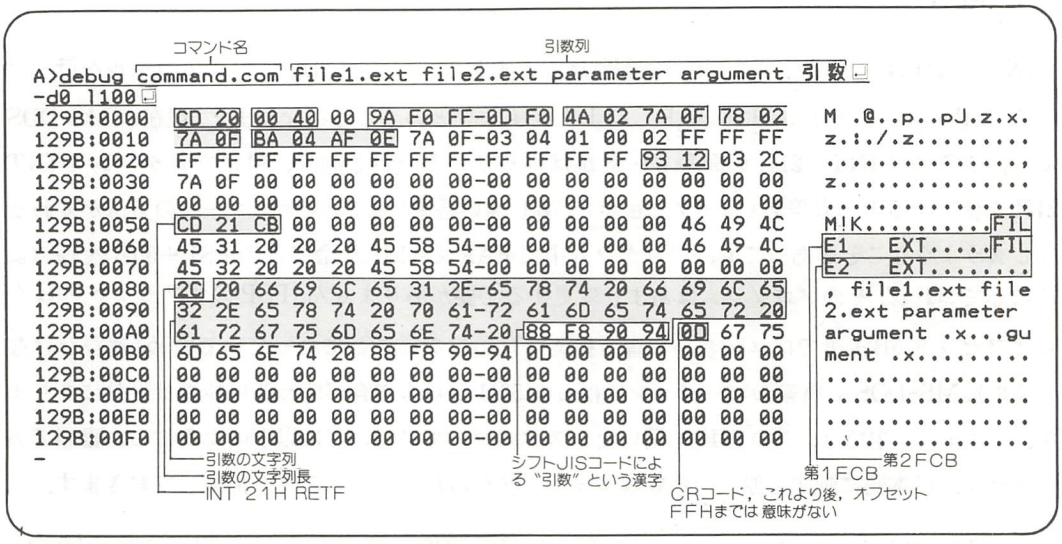


Fig. 3.10　PSPの例

3.3.4　環境

MS-DOSでは，アスキーコードで表現された文字列の最後に1バイトの0(00H)を付加したものを，ASCIZ文字列(ASCII＋ZERO)と呼びます．たとえば，ASCIZ文字列の〝41H, 42H, 43H, 44H, 45H, 00H〟は，文字列〝ABCDE〟を表します．

環境は，単にASCIZ文字列の集まりとして表されます．1つのASCIZ文字列で環境変数(変数名)とその値(文字列)を表し，それがいくつか集まって1つの環境を形成します．環境の最後には，ASCIZ文字列の00Hに加えてもう1つの00Hが置かれています．つまり，環境の占める大きさはどこにも記憶しておらず，その大きさを調べたい場合は最初から最後まですべてスキャン(走査)していかなければなりません．

また，環境はメモリ中ではセグメントの境界から配置されているため(オフセット＝00H)，環境の置かれているアドレスはセグメント・アドレス(1ワード)のみで表すことができます．さきほどのPSPの説明にあったオフセット2CHの環境アドレスとは，このセグメント・アドレスのことを意味しています．

ここでは，DEBUGを使って実際の環境の例をFig. 3.11に示します．パラメータを付けずにDEBUGを起動した場合，Rコマンドで見える4つのセグメント・レジスタはすべて同一の，ま

だ使われていないPSPを指しています．ですから，そのオフセット2CHには環境のセグメント・アドレスがはいっていることになります．

```
A>path a:¥;b:¥bin
A>prompt $g$n$l                          } いくつかの環境を定義する
>A<set computer=PC9801

>A<
>A<set
COMSPEC=A:¥COMMAND.COM
PATH=A:¥;B:¥BIN                          } 現在の状態
PROMPT=$g$n$l
COMPUTER=PC9801

>A<debug
-d0 130        PSPを表示する                          環境のセグメント・アドレス
1804:0000  CD 20 00 40 00 9A EE FE-1D F0 4A 02 FC 14 78 02   M .@..n~.pJ.¦.x.
1804:0010  FC 14 BA 04 33 14 33 14-03 04 01 00 02 FF FF FF   ¦.:.3.3.........
1804:0020  FF FF FF FF FF FF FF FF-FF FF FF FF F6 14 05 2C   ............v..,
-
-d14f6:0        ……環境を見る      環境を構成する1つの要素(ASCIZ文字列)の終わり
14F6:0000  43 4F 4D 53 50 45 43 3D-41 3A 5C 43 4F 4D 4D 41   COMSPEC=A:¥COMMA
14F6:0010  4E 44 2E 43 4F 4D 00 50-41 54 48 3D 41 3A 5C 3B   ND.COM.PATH=A:¥;
14F6:0020  42 3A 5C 42 49 4E 00 50-52 4F 4D 50 54 3D 24 67   B:¥BIN.PROMPT=$g
14F6:0030  24 6E 24 6C 00 43 4F 4D-50 55 54 45 52 3D 50 43   $n$l.COMPUTER=PC
14F6:0040  39 38 30 31 00 00 00 00-00 00 00 00 00 00 00 00   9801............
14F6:0050  5A FC 14 04 2B 00 00 00-00 00 00 00 00 00 00 00   Z¦..+...........
14F6:0060  CD 20 00 40 00 9A F0 FF-0D F0 B0 01 33 14 BD 01   M .@..p..p0.3.=.
14F6:0070  33 14 BA 04 33 14 33 14-03 04 01 00 02 FF FF FF   3.:.3.3.........
-q
>A<■
```

PSPの一部
1つの環境
ASCIZ文字列の終わり
環境の終わりを表す00H
パラグラフの境界に合せるための空き領域，00Hがはいっているとは限らない

Fig. 3.11　環境の例

環境について注意しておかなければならないのは，あるプログラムで使える環境はその親プログラムの環境のコピーであるということです．つまり，あるプログラムから別のプログラムを起動する場合，起動した側のプログラムの環境のコピーが作られ，その環境へのポインタがPSPのオフセット2CHへはいります．ですから，あるプログラム中でいくら環境を変更しても，その親のプログラムの環境は変わりません(Fig. 3.12)．

①実行例

```
A>prompt $n:
A:path a:¥;b:¥bin                        } 環境のセットアップ
A:set var=string

A:set
COMSPEC=A:¥COMMAND.COM
PROMPT=$n:                               } 現在の状態
PATH=A:¥;B:¥BIN
VAR=string
```

```
A:command↵
Command ﾊﾞｰｼﾞｮﾝ 2.11 } 新しいプロセスを起動する

A:set↵
COMSPEC=A:¥COMMAND.COM
PROMPT=$n:
PATH=A:¥;B:¥BIN          } 受け継いだ環境
VAR=string

A:prompt $lMS DOS$g↵
                         } 変更／追加
<MS DOS>set var='NEW STRING'↵
<MS DOS>set new_var=value↵

<MS DOS>set↵
COMSPEC=A:¥COMMAND.COM
PATH=A:¥;B:¥BIN
PROMPT=$lMS DOS$g        } 変更後の環境
VAR='NEW STRING'
NEW_VAR=value

<MS DOS>exit↵ ……もとのプロセスへ戻る

A:set↵
COMSPEC=A:¥COMMAND.COM
PROMPT=$n:               } もとのままである
PATH=A:¥;B:¥BIN
VAR=string

A:■
```

②概念図

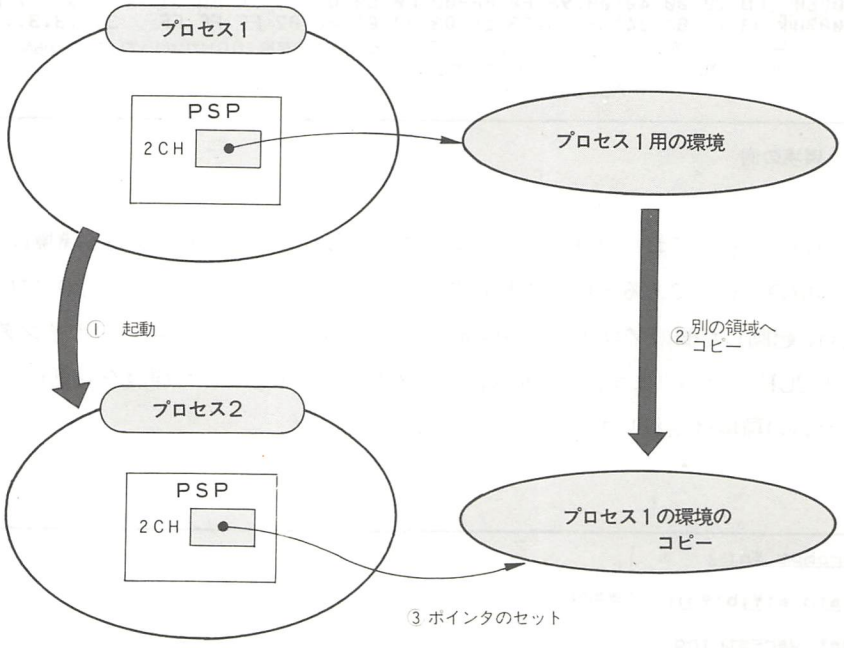


Fig. 3.12　環境のコピー

もう1つ注意しておかなければならないのは，環境の変更についてです．あるプログラムから使える環境は，たいていそのプログラムのPSPの直前にはいっています．つまり，システムコー

ルが新たにプログラムを起動する場合，Fig. 3.13 に示すようにまず空いているメモリの先頭に環境のコピーを作り，次に PSP を作って，最後にプログラム本体をメモリへロードするわけです．

このため，環境自体の総計が小さくなるように変更する分にはかまわないのですが，大きくすることはできません(大きくすると，PSP の領域を壊してしまう)．ですから環境を増やすためには，いったん自分のプログラムの作業領域に環境をコピーしてきておいてから，その後ろへ付け足さなければなりません．というのは，PSP が壊れないようにするためもありますし，環境が 2 つ以上に分断されるのは認められていないからでもあります．

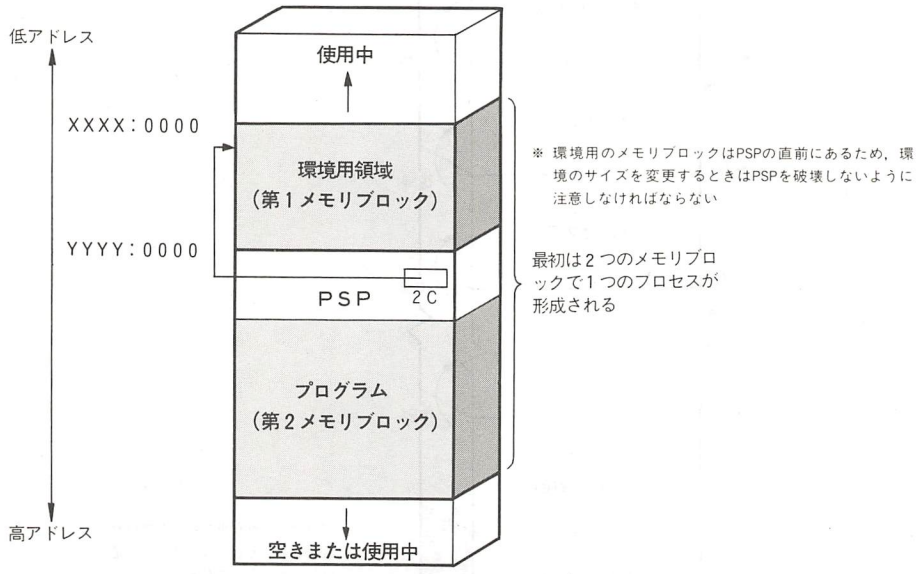


Fig. 3 .13　環境とPSPとプログラムのメモリ配置

3.4　プログラムの実行開始と終了

プログラムの実行開始と終了についてはこれまでも何度か出てきましたが，ここではそれについて詳しく見ていくことにします．

MS-DOS では，メモリ中にロードされて実行されるプログラムのことを〝プロセス〞(process)と呼びます．そして，あるプログラムをロードし，実行させたもとのプログラムを〝親プロセス〞，実行を開始したプログラムのことを〝子プロセス〞といいます．また，子プロセスからさらに子のプロセスを作ることもできるので，Fig. 3.14 のようにメモリの許す限りプロセスを起動することができます．

ここで断っておかなければならないのは，プロセスとは，一般の OS ではマルチタスクのシステムのみで使われる言葉であるということです．マルチタスクとは，1 つ(または多数)の CPU 上で

多数のプログラムを同時に実行させることを指します．もちろん実際はCPUの数より多くは同時には動きませんので，ある一定時間(ms単位ぐらい)ごとに各プロセスを実行し，見かけ上同時に動かしているのです．

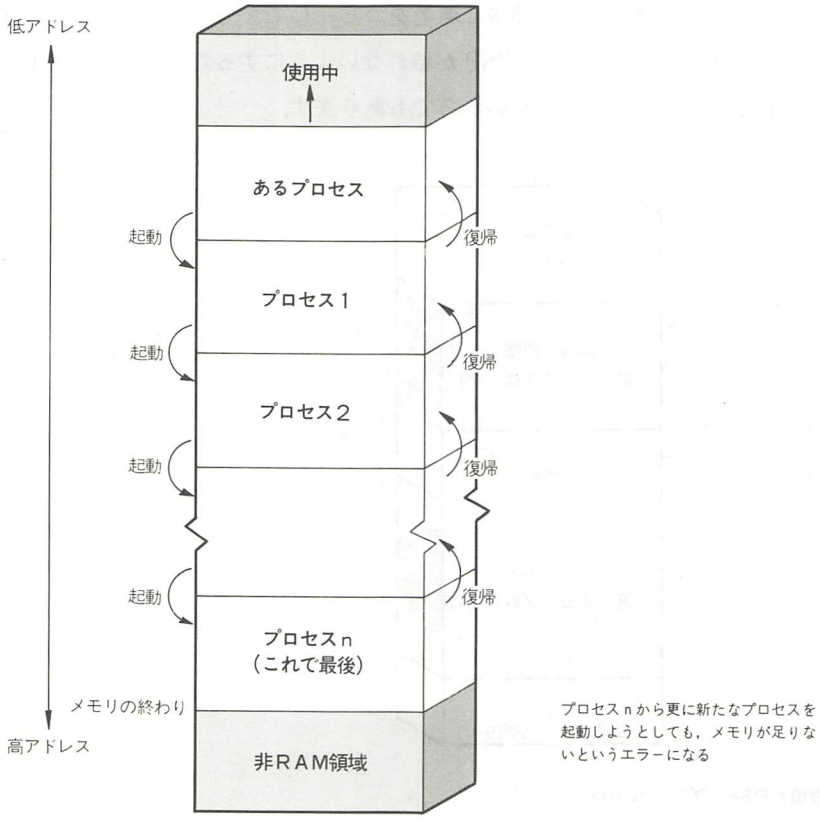


Fig. 3.14　複数プロセスの起動

MS-DOS ver. 2.0は，シングルタスク／シングルユーザーのDOSであるにもかかわらず，あえてプロセスという言葉を使っているのは，将来への伏線なのかもしれません．つまり，MS-DOS ver. 3.0，あるいは4.0なるものができたとすれば，きっとそれはマルチタスクのシステムとなっていることでしょう．

さて，そのプロセスですが，今述べたようにMS-DOSはシングルタスクのDOSであるため，子プロセスを起動した親プロセスは子プロセスの実行が終わるまで待たされることになります．さらにその子プロセスが孫プロセスを起動した場合，孫プロセスの実行が終わるまで子プロセスの実行が止まります．同時に2つ以上のプロセスは動けないのです(Fig. 3.15)．

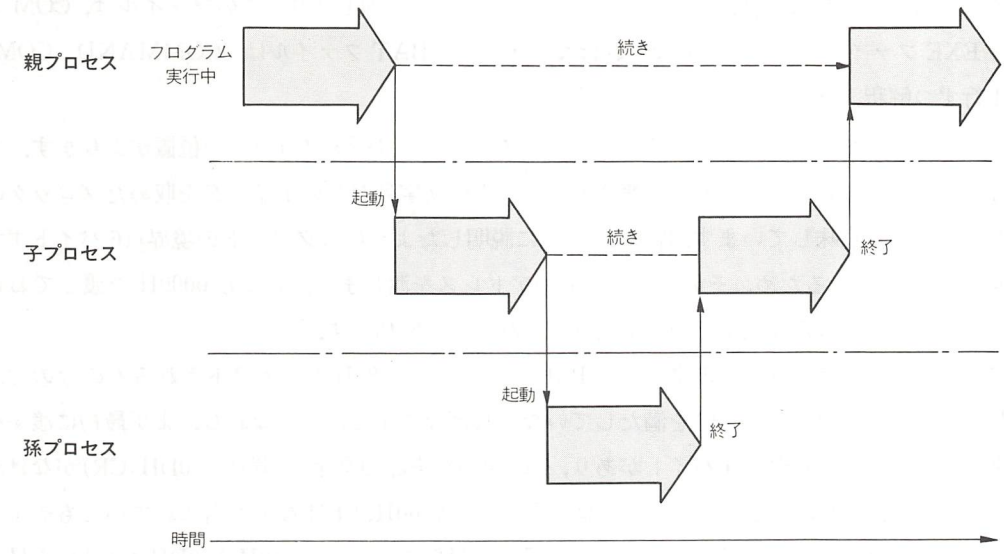


Fig. 3 .15　親プロセス・子プロセス・孫プロセス

MS-DOS では，プロセスの起動や終了用にいくつかのシステムコールが用意されています．

プロセスの起動には，ファンクション 4BH(EXEC システムコール：プログラムのロードと実行)の〝子プロセスの起動〟を用います．パラメータとしては，起動すべきプロセスのはいっているファイルを示すパス名のほかに，環境のはいっているセグメント・アドレス，そしてプロセスへのパラメータを表す文字列が必要です．

プロセスを終了させるためには，システムコールの 00H または 4CH，あるいは INT　20H の3通りの方法があります．いずれの方法を用いても，制御は親プロセスへ戻ります．

それでは，起動と終了の過程について詳しく見ていきましょう．

3.4.1　プロセスの起動

新しいプロセスを起動するためには，EXEC システムコールを使うしか方法がありません．自分のプログラム中で勝手に PSP を作り上げ，プログラムをロードしてジャンプしても，システム側(MS-DOS 側)には同一のプロセス内のこととして認識されます．

EXEC システムコールへ渡すパラメータとしては，まず第一に起動すべきプロセスのはいっているファイルのパス名があります．パス名は，システムコールで用いる場合は，さきほどの環境の説明で出てきた ASCIZ 文字列で表されます．たとえばパス名〝B：¥BIN¥PROG．COM〟は，次のように表されます．

[42, 3A, 5C, 42, 49, 4E, 5C, 50, 52, 4F, 47, 2E, 43, 4F, 4D, 00](16 進)

このASCIZ文字列の先頭のアドレスをシステムコールへ渡します．このファイルは，COMまたはEXEファイルのどちらかでなければなりません．BATファイルは〝COMMAND．COM〟で1行ずつ解釈，実行されています．

システムコールへ渡すほかのパラメータとしては，パラメータブロックの位置があります．これは，環境のアドレスとプロセスへ渡すパラメータの文字列などのポインタを収めたブロックの先頭アドレスを意味しています．環境は，すでに説明したようにセグメントの境界(16バイトずつの境界)からはじまるため，そのセグメントのアドレスを渡します．しかし0000Hを渡しておけば，現在のプロセスの環境がそのまま引き継がれるので便利です．

プロセスへ渡すパラメータの文字列はPSPのオフセット80Hからセットされるものなので，正しくそのフォーマット(形式)を満たしていなければなりません．すなわち，まず最初に渡すべき文字列の長さ(0～126)の1バイトがあり，その次から実際の文字列，最後に0DH(CR)がなければなりません．また，文字列中にコントロールコードや00H, FFHなどが含まれていてもチェックは行われません．なお，渡すパラメータの長さは最低で2バイト(00Hと0DHだけ)から最大128バイトです．

今述べたパラメータはPSPのオフセット80Hからのためのものでしたが，システムコールを呼ぶためには，もう2つ，PSPの5CHと6CHのFCB用のパラメータも必要です．これらは，そのままFCBにはいるフォーマットをしていなければなりません．つまり，まず最初のバイトにドライブ番号を入れ，2バイト目からは11文字(8+3文字)のファイル名がはいっているものでなければならないのです．そしてこれらはさきほどのPSPの実例(Fig．3．10)を見てもわかるように，80Hに置くべきパラメータの第1と第2のパラメータでなければなりません．なお，第1と第2のパラメータがファイル名でない場合，ただの空文字列を渡してもよいでしょう．

3.4.2　新しいプロセスの実行

プロセスの実行開始に先立って，まずPSPが作られます．PSPの中で，オフセット00Hからの2バイト(CDH, 20H)とオフセット50Hの3バイト(CDH, 21H, CBH)はどのPSPでも共通ですが，ほかの領域は状況によって変化します．

オフセット02Hからの使用可能なメモリの上限アドレスとオフセット06Hからの2バイトは使用可能なメモリの量によって，決定されます．

オフセット0AHからは，2ワードのアドレスが3つはいっていますが，これらはPSP作成時に8086CPUの割込みベクトル領域のINT　22H～INT　24Hのテーブルからコピーされるものです．

このコピーに先立って，INT　22H(終了用アドレス)にはEXECシステムコールの命令の次のアドレスが設定されています．PSPのオフセット0AHへコピーされるのは，INT　22Hのベクトルにアドレスがセットされてからです．

MS-DOS ver. 1.25 では，INT　22H のベクトル領域はユーザーがプログラム中で明示的に書き換えなければなりませんでしたが，その代わり子プロセスのプロセスの終了をどこへでも好きなアドレスに設定することができました．しかし ver. 2.0 では，プロセスの終了時には親プロセス中の EXEC システムコールの次の命令へ制御が移ることになります．

INT　23H と INT　24H のベクトルは，PSP 作成時にオフセット 0EH～15H へコピーされます．ベクトル領域へは，INT　22H のように何か書き込まれるということはありません．ですから，CTRL-C による中断と致命的エラーによる 2 つのルーチンは，ユーザーが望むならば各プロセスごとに用意することができます．なお，この点に関しては再度詳しく解説します．

以上 3 つのベクトルのコピーが終わると，次に環境アドレスの設定が行われます．実際には，まず環境のコピーが作られ，そのすぐ後ろに PSP が置かれます．ですから，PSP のオフセット 2CH～2DH へは PSP の直前にある環境のセグメント・アドレスがはいります．

PSP のセグメント 5CH 以降は，単に EXEC システムコールへ渡されたパラメータのコピーです．

PSP には，今まで述べたもののほかにもいくつかの特別な意味を持った値がはいっています．しかしそれらは，たいていの場合ユーザーにとって意味がなく，またその内容などについては，将来変更される可能性が十分にあるので，ここではそれらについてはいっさい言及しません．興味のある方は，それらがいったい何を意味しているのか探究してみるのもおもしろいでしょう．ただしユーザーは，PSP の 5CH 未満の部分についてはあまりさわらない方が無難です．大切なプログラムやデータ，ファイル，さらにはハードウェアが壊れてしまうかもしれません．

ここまでで，PSP のセットアップについてはひととおり説明してきたわけですが，最後にそれを簡単にまとめておきましょう．

(1)親プロセスの環境を子プロセス用のメモリへコピーする
(2) PSP 用のメモリ領域に(1)のアドレスをセットする
(3) INT　22H のベクトルに終了アドレスをセットする
(4) INT　22H～INT　24H のベクトルを PSP へコピーする
(5)メモリ容量に関連する領域の値をセットする
(6) 5CH, 6CH の FCB をセットする
(7) 80H からのパラメータ用領域をセットする
(8)そのほか，必要な領域に値をセットする

これで PSP の準備ができましたので，次に〝MSDOS．SYS〟はプログラム本体をロードし，実行を開始します．もちろん，単にジャンプするだけでなく，後述する〝MSDOS．SYS〟内のファイル・ハンドルやメモリ管理用の情報などをセットしてからプロセスへ制御が移ります．そして，子プロセスの方でプロセスの終了用のシステムコールが使われると，制御が親プロセスの方へ戻ってきます．

3.4.3　プロセスの終了

終了の手続きにおいて行われる処理はプロセスの起動前の状態に戻すことが目的ですので，およそ次の処理が行われます．

(1)子プロセスでオープンしたファイルをすべてクローズする
(2) PSP のオフセット 0AH からの 3 つのダブルワードのポインタを，INT　22H～INT 24H のベクトルの領域へ戻す
(3)環境や PSP，プログラム自身で占めていたメモリ領域をすべて解放し，その他のプログラムで使えるようにする
(4) INT　22H のベクトル，または PSP のオフセット 0AH にあったベクトルへロングジャンプする

もちろん，(4)の処理が行われる前に，"MSDOS．SYS" 内の現在のプロセスについて持っている情報はもとのプロセスのものに戻されます．

3.5　プログラムのチェインと中断

本節では，プログラムの起動と終了に伴う処理において，PSP にコピーされたりもとに戻されたりする 3 つのベクトルの用途について説明していきます．

3.5.1　INT　22H ベクトル

INT　22H ベクトルはプロセスの終了アドレスを保持しており，EXEC システムコールを使用するとその値の設定およびもとの値への回復が自動的に行われます．つまり，EXEC した時点で戻りアドレスが PSP と INT　22H のベクトル領域へ格納され，終了時には逆にもとに戻すので，ちょうど機械語における CALL と RET のような処理を行います．ですから，EXEC システムコールを使用すれば，あるプロセスの中から次のプロセスを呼び出し，その中からまた次のプロセスを呼ぶということをしても，それぞれのプロセス中で正しく終了すれば 1 つずつもとのプロセスに戻ります．

多くの場合，いちばん親となるプロセスは "COMMAND．COM" であるため，ユーザーが実行を指示したプロセスはその実行が終わると "COMMAND．COM" に戻ります．また，"COMMAND．COM"自身も 1 つのファイルであるため，これをコマンドとして(COMMAND．COM から)起動できます．このような場合，MS-DOS の内部コマンドの EXIT コマンドを実行すると，新たに起動された方の "COMMAND．COM" の実行が終了し，制御は最初の "COMMAND．COM" へ戻ります．もちろんこのような場合でも，2 番目の "COMMAND．COM" からどのコマンドも解釈，実行することができます．

3.5.2　INT　23H, INT　24H ベクトル

INT　23H と INT　24H の 2 つのベクトルも，プロセス起動時に 8086CPU の割込みベクトル領域から PSP にコピーされ，終了時にもとに戻されます．そのため，プロセスの実行中にこのベクトルを書き換えても，終了時にはプロセスの起動前の状態に戻ります．このため，各プロセスではそれぞれ独自の CTRL-C 処理ルーチンや致命的エラー処理ルーチンを持つことができます (Fig. 3.16)．

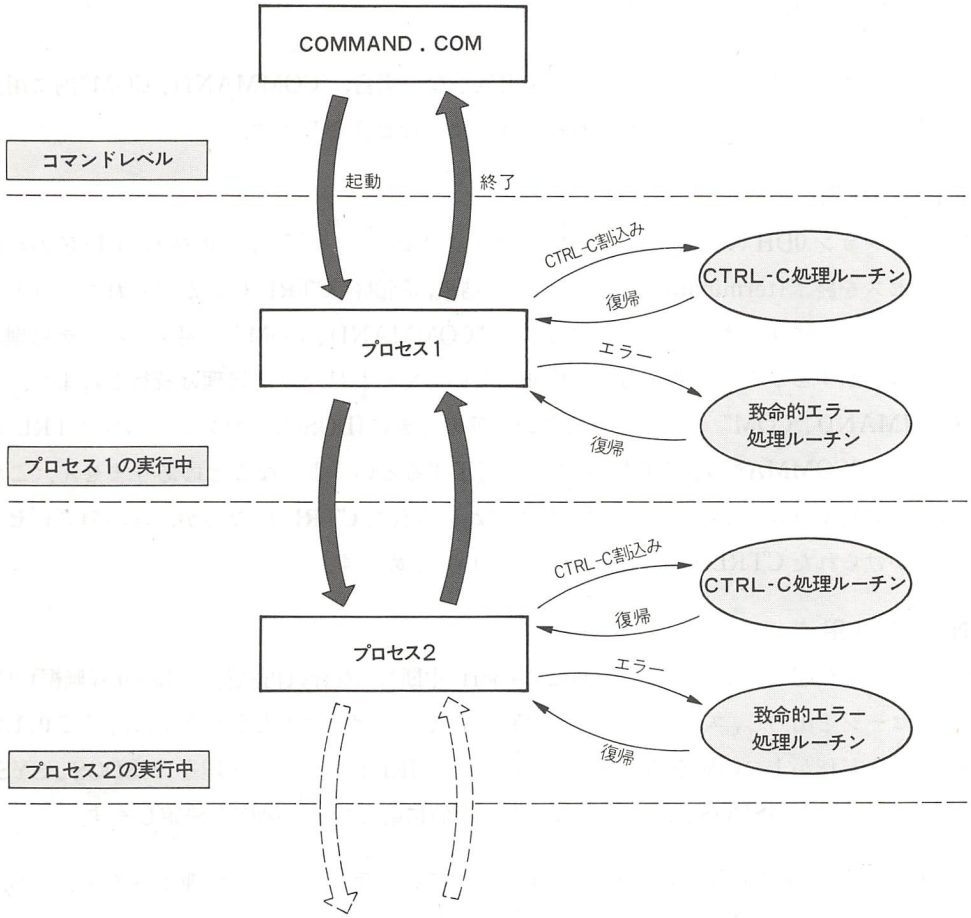


Fig. 3．16　CTRL-Cと致命的エラー処理ルーチン

このように，各プロセスで処理ルーチンを用意するためには割込みベクトルの値を書き換えておきます．実際の処理としては，ファンクション 25H のシステムコール(割込みベクトルの設定)を使用して，次のような命令を各プロセスの最初の方で行っておけばよいでしょう．

● CTRL-C の処理ルーチン

```
MOV  DX, OFFSET  CTRL_C
MOV  AX, 2523H
INT  21H
```

●致命的エラーの処理ルーチン

```
MOV  DX, OFFSET  FATAL_ERROR
MOV  AX, 2524H
INT  21H
```

なお，各プロセスでこの2つのルーチンを用意しない場合，〝COMMAND．COM〟内に用意されている，デフォルトのルーチンがそれぞれ次のような動作をします．

○ INT　23H 用

ファンクション 0DH のシステムコール(ディスクリセット)を行い，その後 CTRL-C のかけられたプロセスを終了(terminate)させます．その結果，制御は CTRL-C のかけられたプロセスの親プロセスへ戻ります．さらに，親プロセスが〝COMMAND．COM〟の場合，バッチ処理の実行中ならば中断するか否かを問い合せ，中断しないのであればバッチ処理は続行されます．

〝COMMAND．COM〟がコマンド待ちの状態のときに(DOS のコマンドレベル)CTRL-C を入力しても，〝COMMAND．COM〟の実行が終了するというようなことはありません．これは〝COMMAND．COM〟が，自分自身の実行中にかけられた CTRL-C なのか，ほかのプロセスの実行中にかけられた CTRL-C なのかを区別しているためです．

○ INT　24H 用

致命的エラーが起こったときに，画面に〝Abort(中断)，Retry(再試行)，Ignore(無視)？〟というメッセージを出力し，ユーザーの選択を待ちます．ユーザーの入力した文字に応じて 0,1,2 の3つのうちのどれか1つの値を AL レジスタに入れて IRET を行い，処理を〝MSDOS．SYS〟に戻します．この後〝MSDOS．SYS〟では，3つの値に応じてその動作を決定します．

この2つが〝COMMAND．COM〟内に用意されているデフォルトの処理ルーチンですが，自分でこれらのルーチンを作る際に必要となることについて以下に述べておきます．

○ INT　23H 処理ルーチン

INT　23H 処理ルーチンは，〝MSDOS．SYS〟がシステムコール(INT　21H)のファンクション 01H～0CH(コンソール関係の入出力処理)の処理中にユーザーが CTRL-C を押すと呼び出されます．BREAK スイッチが ON になっているとすべてのシステムコールについてブレークチェックが行われ，CTRL-C が押されたときに，この INT　23H 処理ルーチンが呼び出されます．こ

のとき，すべてのレジスタの値がシステムコールを行ったときの値に戻されてから，INT　23H 命令によって処理ルーチンが呼び出されるため，この処理ルーチン内で全レジスタの内容を保存しておけば，IRET 命令によって処理を続行することができます(〝MSDOS．SYS〞内のシステムコール処理ルーチンから処理が再開される．当然のことながら，その際ブレークチェックは行われず，システムコール処理中に押された CTRL-C は次回のシステムコールのときにチェックされる)．

INT　23H 処理ルーチンの呼出しは，〝MSDOS．SYS〞がシステムコールの処理を開始する前に行われるので，必要とあらば処理ルーチン中でシステムコールを利用してもかまいませんが，ここでシスムコールを用いるとさらに CTRL-C によるブレークが起こる可能性があるため，そのことも考慮して処理ルーチンを作っておかなければなりません．

○ INT　24H 処理ルーチン

〝MSDOS．SYS〞がシステムコールを処理中に Table 3.2 に示す致命的エラーと呼ばれるものが発生した場合，制御は INT　24H のベクトルのアドレスへ移行します．

Table 3．2　致命的エラーとエラー番号

番号	エラー内容
0	ライトプロテクトされているディスクへの書込み
1	存在しないユニット番号
2	ドライブの準備ができていない
3	コマンド番号が間違っている
4	データエラー(CRCエラーなど)
5	コマンドパケットの長さが間違っている
6	シークエラー
7	メディアタイプが間違っている
8	セクタが見つからない
9	プリンタの用紙切れ
10	書込みエラー
11	読出しエラー
12	その他のエラー

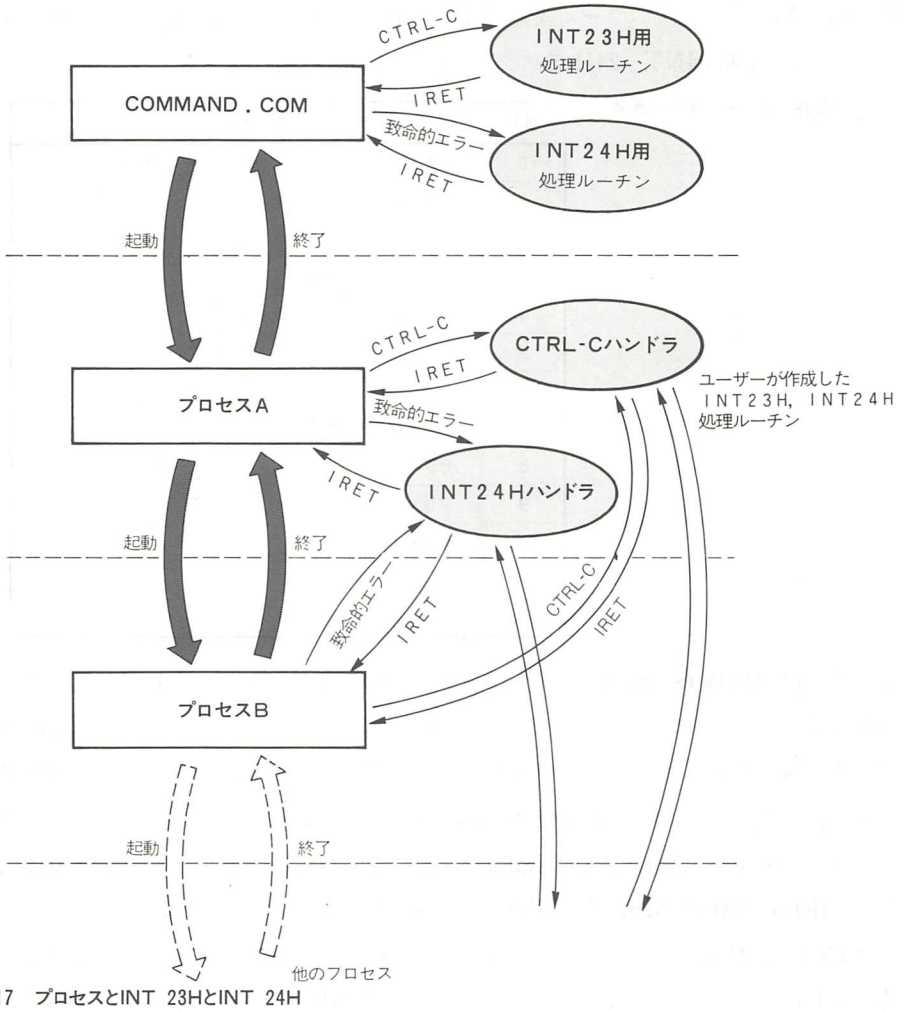
このルーチンは〝MSDOS．SYS〞の中から呼び出され，処理が終了すると IRET 命令で戻らなければなりません．このルーチン内でのシステムコールは，ファンクション 01H～0CH だけは使えますが，そのほかのシステムコールを使用すると〝MSDOS．SYS〞内のデータ領域が破壊されてしまいます．このルーチンで行える処理は，エラーコードに従ってエラーメッセージを画面に出力し，その後の動作を Abort, Retry, Ignore の 3 つの中から 1 つ選び，その情報(0,1,2 のどれか) を持って IRET 命令で〝MSDOS．SYS〞に戻るということです．Abort を選ぶと，プログラムの実行は INT　22H のベクトルの値(プログラムの終了アドレス)へ移行します．なお，このときの画面の出力はシステムコールのファンクション 01H～0CH を用いて出力するので，ここで

CTRL-Cを入力すると制御はINT 23Hの処理ルーチンへ移ります．このことも考慮して，その際のINT　23H処理ルーチンは，ファンクション01H～0CH以外のシステムコールを避けて使用しなければなりません(ただし，ファンクション4CH：ターミネート・システムコールは除く)．

またINT　24H処理ルーチンから，"MSDOS．SYS"に戻らずにそのままユーザーのプログラムへ戻るには，スタック上に積まれている各レジスタ(システムコールを呼んだ時点での各レジスタがスタック上に積まれている)の内容をもとに戻して(必要がなければ戻さなくてもよい)IRET命令を使います．すると，もとのアドレス(システムコールINT　21Hの次のアドレス)へ制御が移ります．

○エラー処理ルーチンとプロセス

以上でプログラムのチェイン(呼出し／終了)と中断に関する説明を終えますが，これらとプロセスとの関係について少々説明を付け加えておきます．



Fig．3．17　プロセスとINT 23HとINT 24H

一般に，ユーザーがINT　23Hや INT　24Hのルーチンを作成する場合，その作成したルーチンが別の(子)プロセスから呼び出される可能性があります．

Fig. 3.17でいえば，プロセスAで定義されたINT　23HとINT　24Hの処理ルーチンは，プロセスAを親(祖先)に持つすべてのプロセスから共有されます．このとき〝MSDOS．SYS〟は，プロセスの環境を保ったままそれぞれの処理ルーチンへ制御を移します．たとえば，プロセスBを実行中にCTRL-C割込みが起こると，プロセスBの実行環境(ファイルの使用状況やメモリの使用状態などの総称)のまま，プロセスAで定義されたINT処理ルーチンに制御が移ります．ですから，たとえ処理ルーチンの中で単にターミネート・システムコール（ファンクション4CH）を用いても，制御は〝COMMAND．COM〟ではなくプロセスAに移行します(この場合，プロセスBへのEXECシステムコールの次の命令へ戻る)．

3.6　FCBとファイル・ハンドル

FCB(File Control Block)とファイル・ハンドル(File Handle)は，ともにユーザーがアセンブラレベル(機械語レベル)でシステムコールを扱う上で非常に大切な概念です．

MS-DOS ver. 1.25やCP／Mでは，機械語レベルでファイルを扱おうとすると(ファイルの読み書き，削除，作成など)FCBはかならず必要となるものでしたが，MS-DOS ver. 2.0では，ファイル・ハンドルさえ知っていればファイルが扱えます．しかし，FCBを完全に無視するわけにはいかないため，まずFCBについて説明した後，ファイル・ハンドルについて解説します．

3.6.1　基本FCB

FCBとは，File Control Blockの名が示すように，ファイルを扱う際に必要となる種々の情報を格納しておくための領域のことであり，ファイルを1つ使うごとにユーザー側で1つずつ用意する必要があります(Fig. 3.18)．

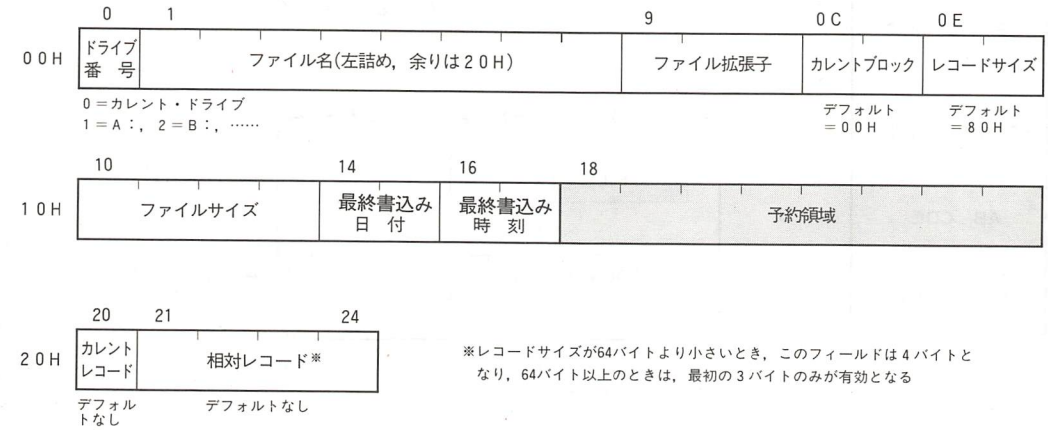


Fig. 3.18　FCB

図の中にシステム側で使用する領域があるということからもわかるように，FCBというのは，その目的のほとんどがシステム(MSDOS．SYS)がファイルをアクセスするときに必要となる情報を入れておく領域です．CP／Mをベースにした MS-DOSでは，CP／M同様FCBを用いてファイルを扱います．CP／Mにしろ MS-DOS ver. 1.25にしろ，FCBを使うのが最良の方法というわけではなく，これは以前からの慣例でそうなっているようです．しかし，UNIXの考えを取り入れている MS-DOS ver. 2.0では，ファイルはファイル・ハンドルで扱うことになっています(詳細は後述)．

FCBの各フィールドの説明に移る前に，まずFCBの使用目的について説明しておきます．FCBは，その使用目的によって2種類に分けることができます．1つはいうまでもなくファイルのオープンやクローズ，読出し，書込みなどに使われますが，それとは違った用途としてファイルの検索や削除などがあります．両者の違いについてまとめると，前者のFCBは1つのファイルを対象としているのに対し(ファイル1つごとにFCBが1つ必要)，後者の場合1つのFCBで複数のファイルを対象とすることができます．ファイル名にワイルドカード(〝？〟は任意の1文字に対応する．〝＊〟はユーザープログラムまたはシステムコールによって〝？〟に展開しなければならない)を含ませることにより，複数のファイルが処理の対象となります(Fig. 3.19)．

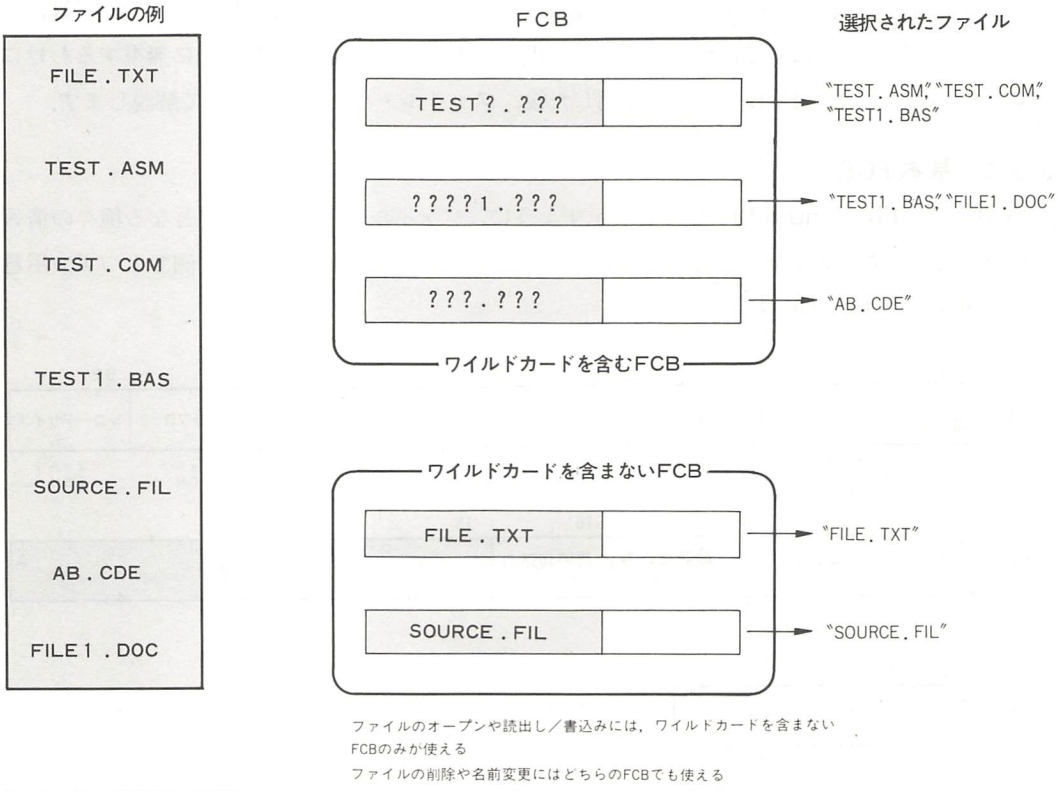


Fig. 3．19　2種類のFCB

この2種類とは別に，FCBはFig. 3.20に示すような，〝オープンされているFCB〟と〝オープンされていないFCB〟の2つに分けることができます．

まず最初に，プログラムでFCB中にファイル名をセットしますが，この段階でのFCBは，〝オープンされていないFCB〟と呼ばれます．このFCBは，さきほどの2種類のFCBのうち，どちらであってもかまいません．つまり，ワイルドカードが含まれていてもいなくてもよいのです．〝オープンされていないFCB〟は，ファイルの検索や削除に使うことができます．

〝オープンされていないFCB〟がワイルドカードを含んでおらず，かつそのファイルが存在するならば，システムコールのファイルのオープン（オープン・システムコール：ファンクション0FH)によってオープンされているFCBにすることができます．また，たとえファイルが存在していなくても(存在していてもよいが)，システムコールのファイルの作成(ファンクション16H)により，オープンされているFCBを作ることができます．ただし，やはりFCB中にはワイルドカードは許されません．

オープンされているFCBを作るには，FCBにドライブ番号とファイル名をセットしてオープン・システムコールをコールします．ドライブ番号はA＝1，B＝2，……となりますが，0番にしておくとオープン・システムコールによりカレント・ドライブ番号にセットし直されます．

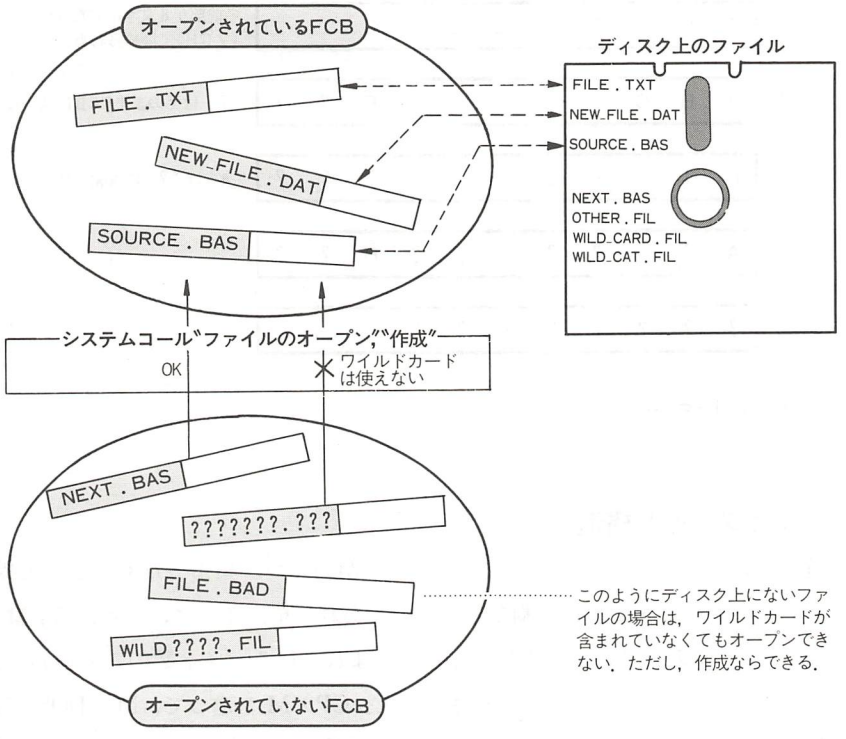


Fig. 3 .20　オープンされているFCBとオープンされていないFCB

ファイル名と拡張子は，それぞれ8および3文字までのアスキー文字が左詰めで入れられますが，各欄とも8または3文字に満たないときは余った右側の部分にスペースのアスキーコード(20H)を入れておかなければなりません．ドライブ名とファイル名，拡張子の組から(一般型は[ドライブ名：]ファイル名[．拡張子])FCBの中の3つの欄を埋めるプログラムを作るのはかなりめんどうですが，よく使われるので，すでにシステムコール(ファンクション29H)として用意されています．

3つのフィールドを正しくセットした後，オープン・システムコールを用いてFCBをオープンすると，FCBのほかのフィールドが意味を持つようになります．これらのフィールドは，オープンやクローズ，読出し，書込みのシステムコールによって順次変更や管理が行われていきます．

なおFig. 3.21のような，ワイルドカード(〝?〟や〝*〟)を含んだファイル名もシステムコール(ファンクション29H)によってFCBにセットすることができます．

しかしこれらのFCBは，ファイルの検索や削除，名前変更(rename)にしか使うことができず，オープンすることはできません．

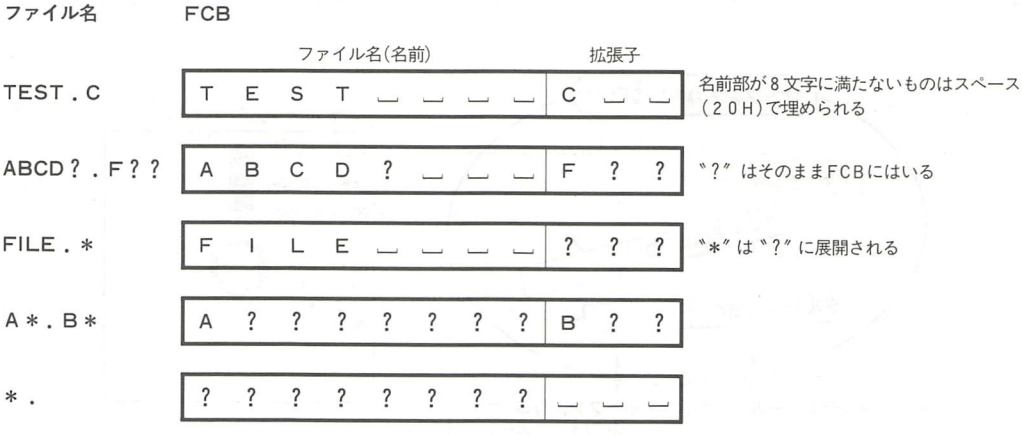


Fig. 3.21　ワイルドカードとFCB

3.6.2　FCBとファイル構造

MS-DOSでは，ファイルは単に1バイトのデータがいくつか集まったものとして扱われ，Fig. 3.22に示すように1バイトごとに分割できるものであれば構造や長さ，内容，名前はいっさい関知しません．ファイルの長さとはそのファイルに含まれているバイト数のことをいい，ファイルをクローズしたときにディレクトリに記録されます．CP／Mと違って，MS-DOSではファイルの長さは1バイト単位(CP／Mは128バイト単位)で管理できるため，ファイルの余った部分(128バイトに満たない端数)について意識する必要はありません．

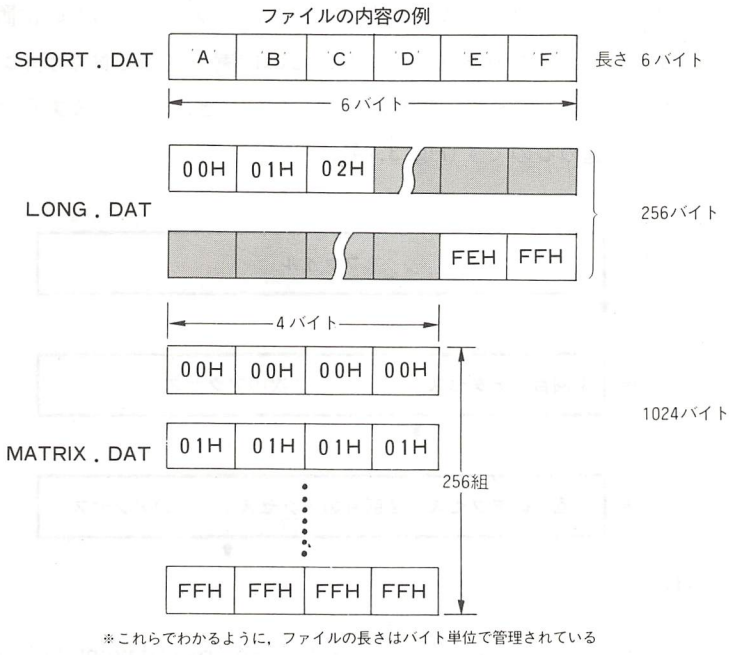


Fig. 3 .22　ファイルの長さ

MS-DOS ではファイルは 1G バイト（2^{30} バイト）まで扱えることになっているため，ファイルのサイズを表すには 4 バイト必要になります．これが FCB のオフセット 10H にはいっている〝ファイルの長さ〟です．

通常，ファイルは先頭から読み書きされますが，このようなファイル・アクセスの方法をシーケンシャル・アクセスと呼びます．これは最も基本的で多用されるファイルのアクセス方法です．この方法ではファイルの読み書きは，前回アクセスした部分の次から行うことになっています．このとき，一度のアクセスで何バイトの読み書きを行うかはプログラムで指定します（Fig. 3.23）．

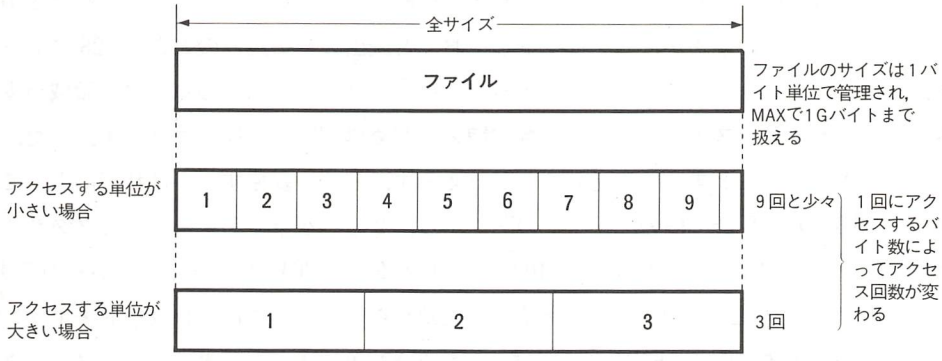


Fig. 3 .23　ファイルのアクセス

このようなシーケンシャル・アクセスを実現するには，次にファイルのどの位置から読み書きするかを覚えていればよいことになります．つまり，この位置を覚えておくポインタをファイルのオープン時にファイルの先頭(オフセット0)にセットしておき，アクセスするたびにそのアクセスした量だけポインタを進めるのです(Fig. 3.24)．

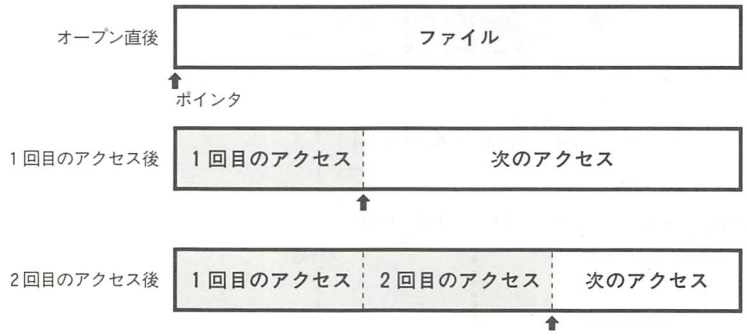


Fig. 3 .24　ファイルとポインタ

MS-DOSではこのポインタのことを，レコード・ポインタ(record pointer)と呼びます．レコードとは，単に1つの記録の単位であり，CP／Mのように1レコードが128バイトというように固定されていません．

ファイルは最大1Gバイトにもなり得るため，レコード・ポインタは4バイト必要となりますが，MS-DOSでは，FCBのカレントブロック，カレントレコード，レコードサイズの3つのパラメータを用いてレコード・ポインタを作ります．

レコードサイズ(2バイト)とは，1回の読出しまたは書込みのシステムコールで何バイト扱うかということを設定するもので，FCBをオープンすると，デフォルトで128バイトに設定されます．この1回のアクセスで扱うデータをレコード(record)と呼びますが，デフォルトで128バイトに設定されているのはCP／Mとの互換性を保つためです．たとえば一度に1バイトずつアクセスしたければレコードサイズを1にしておき，32Kバイトずつアクセスしたければ32768をセットしておけばよいのですが，0をセットしておくと読出しや書込みのときに128にセットされます．一般に，レコードサイズを小さくするとユーザープログラム内のバッファ領域が少なくて済みますが，その分システムコールの回数が増えるため処理速度が遅くなります．また，レコードサイズを大きくするとまったく逆のことがいえます．どこで妥協するかは状況によって異なりますが，"MSDOS．SYS"内のバッファの関係から，2のべき乗でディスクのセクタサイズより大きくしておくのがよいと思われます(1024バイトあるいは2048バイトぐらいが妥当です)．

カレントブロック(2バイト)は，ファイルの先頭から何ブロック目にレコード・ポインタが位置しているかを示すもので，FCBをオープンしたときに0にセットされます．1ブロックはつねに128レコードの集まりであるので，1ブロックの総バイト数はレコードサイズ×128(バイト)と

いうことになります．ブロックはファイルを構成する１つの単位ですが，あまり重要な役割はありません．単にCP／Mのエクステントと対応するために存在するようです．

カレントレコード(１バイト)は，カレントブロックにより示される１ブロック128レコードのうち，次にアクセスされるのはどのレコードであるか(0～127)を保持しています．

以上３つのフィールドの関係をFig. 3.25に示します．

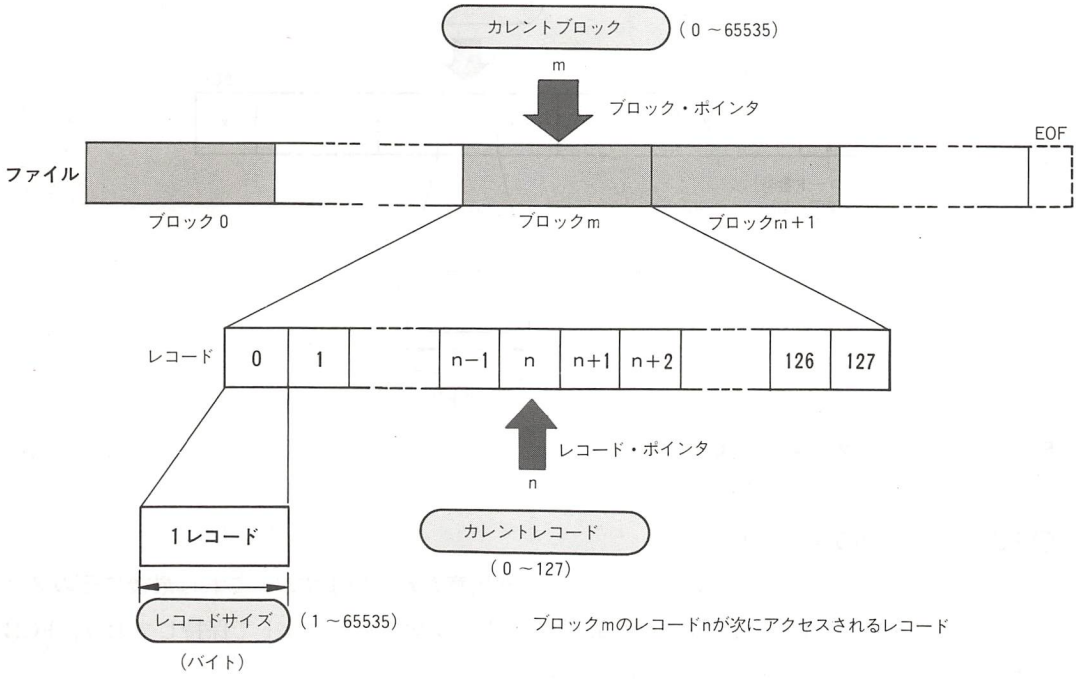


Fig. 3 .25　ファイルのシーケンシャル・アクセス

これらのポインタは，ファイルの読出し／書込み動作に伴って順次更新されます．１レコードをアクセスするごとにカレントレコードが１個増え，127を超えると今度はカレントブロックが１つ増えてレコード・ポインタは０になります．ファイルの終わり(End Of File)を超えて読出しを行おうとすると，１レコードに満たない部分（1～レコードサイズ－1バイト）には０が詰められたものがメモリに読み込まれます．EOFを超えて書き込むと，ディスク上に新しい領域が取られてファイルが延長されます．

これまではシーケンシャル・アクセスの方法について説明してきましたが，ファイルのアクセスにはもう１つランダム・アクセスと呼ばれる方法があります．これは，ファイルを最初から順にアクセスするのではなく，ファイル中の任意のレコードを任意にアクセスする方法です．この方法でファイルをアクセスするためには，FCBの〝レコードサイズ〟と〝相対レコード〟の２つのフィールドを用いてレコードの位置を指定します．レコードサイズは，シーケンシャル・アクセスと同様１レコードの占めるバイト数を保持しており，相対レコードは，ファイルの先頭から

何レコード目かということをセットしておくためのものです．シーケンシャル・アクセスの場合と異なるのは，相対レコードは4バイトあるためこれ1つでファイルの望むところへアクセスできるということです(シーケンシャル・アクセスの場合は，カレントブロックとカレントレコードの2つが必要)(Fig. 3.26)．

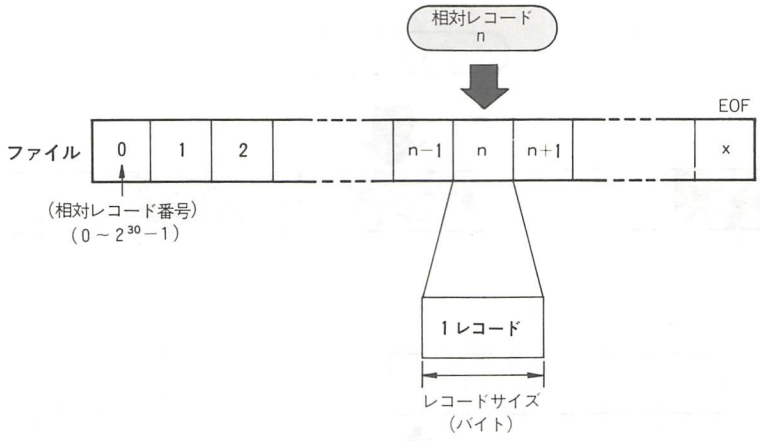


Fig. 3.26　ファイルのランダム・アクセス

○ FCB のその他のフィールド

日付と時刻に関して2バイトずつのフィールドが用意されていますが，これは最後にそのファイルに書込みを行った日付と時刻を Fig. 3.27 に示すようなフォーマットで保持しており，FCB がクローズされた際ディスク上に書き込まれます．

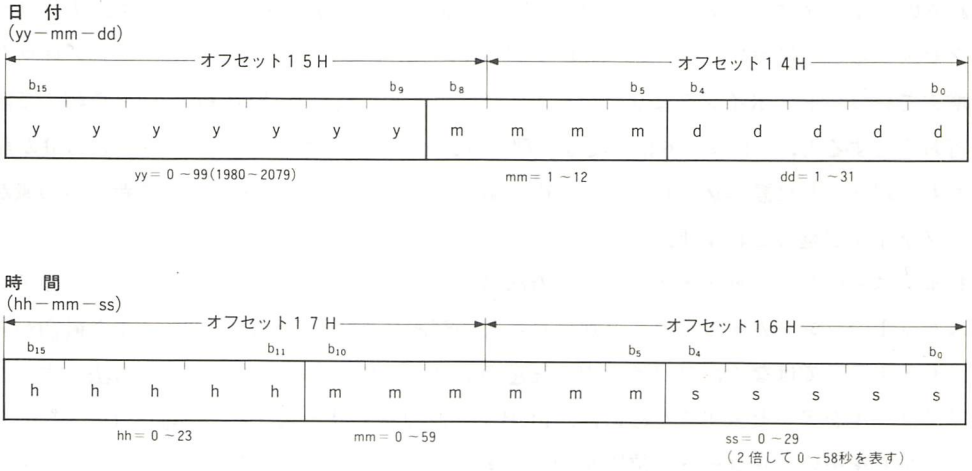


Fig. 3.27　時間のフォーマット

FCBの予約域には，"MSDOS．SYS"がこのファイルを管理するための情報がはいっていますが，このような領域をFCBに持たせてユーザーのプログラムの中へ置くことにより，扱うファイル数の制限をなくすことができます．

拡張FCB

拡張FCBとは基本FCBの前に付加する7バイトの拡張部のことでありFig. 3.28に示すようなフィールドを持っています．この拡張FCBを使うことにより，通常のファイルとは違う属性を持ったファイルのアクセスを可能にします．

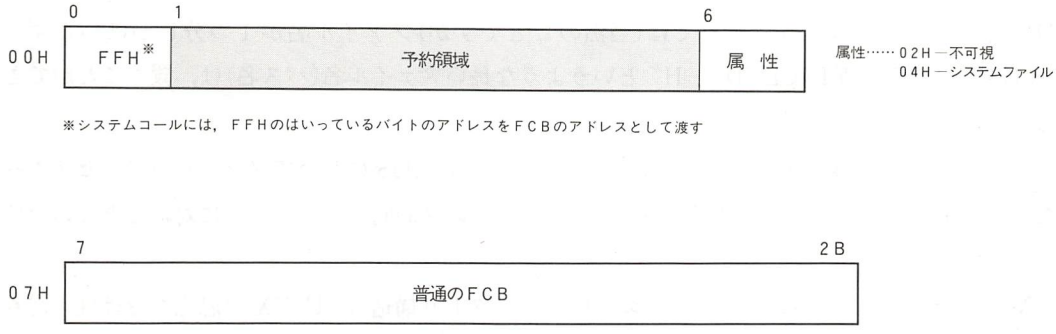


Fig. 3．28　拡張FCB

購入した時点でのMS-DOSのディスクでは，"IO．SYS"とか"MSDOS．SYS"はDIRコマンドでは表示されませんが，それは，これらのファイルが「システムファイル」とか「隠されたファイル(hidden file)」などの属性を持っているためです．拡張FCBを用いると，これらのファイルに対してはアクセス(読出し／書込み)ができるようになりますが，基本FCBでアクセスしようとしても，"ファイルが見つからない"というエラーになってしまいます．

デバイス名とFCB

MS-DOSでは，CP／Mとは違ってデバイスもファイルと同格にオープンやクローズができることが，その特徴の1つとなっています．

デバイスをオープンするには，ファイルと同じようにFCBのファイル名のフィールドへデバイス名(通常3文字で":"を含めてはならない)を入れ，フィールドの余った部分に20Hを入れておいてファイルと同じようにオープンします．後は単に読出し／書込みを行えば，そのデバイスとデータがやり取りできます．注意しなければならないのは，PRNからデータを読むようなことをしてはいけないということです．こういうことをすると，動作が保証されません(通常リブートしなければならなくなる)．

3.6.3　ファイル・ハンドル

FCBを使ってファイルをアクセスするには，まずFCBにファイル名をセットし，それをオープンして，レコードサイズやカレントブロック，カレントレコードを正しくセットしてからファイルの読出し／書込みを行うという手続きが必要です．ランダム・アクセスの場合は，レコードサイズと相対レコードをセットします．これらの手続きは，慣れてしまえば何ともないのでしょうが，名前のセットはともかく，ブロックやレコードについてまで管理しなければならないのでは，やはりめんどうです．

また，FCBにはもう1つ重大な欠点があります．それは，階層的ディレクトリ構造を扱うことができないということです．というのはFCBのファイル名のフィールドは，拡張子も含めて合計11文字分しかありません．これでは(当然のことですが)ファイル名が1つ分しかはいらず，〝¥USR¥LOCAL¥INCLUDE．H〟というような長いファイル名(パス名)は，置くことができないのです．

元来FCBは，階層的ディレクトリ構造になっていないDOSにおいてファイルをアクセスするために使用されていた技法なので，MS-DOS ver. 2.0の階層的ディレクトリに対応できないのは当然です．

MS-DOS ver. 2.0で採用されている階層的ディレクトリ構造は，UNIXの思想を受け継いだものなので，ファイル・アクセスの方法もUNIXのものを取り入れた方が自然です．というわけで，MS-DOS ver. 2.0にはUNIX(XENIX)コンパチブルなシステムコールが新たに追加されました．これは，追加であって置き換えではないので，従来のFCBを使ったプログラムでも問題なく動かすことができます．

UNIXではファイルをアクセスするのにファイル記述子(file descriptor)を用いますが，MS-DOS ver. 2.0では，同じものをファイル・ハンドル(file handle)と呼んでおり，ファイルをアクセスするにはこのファイル・ハンドルをシステムコールに渡すようにします．

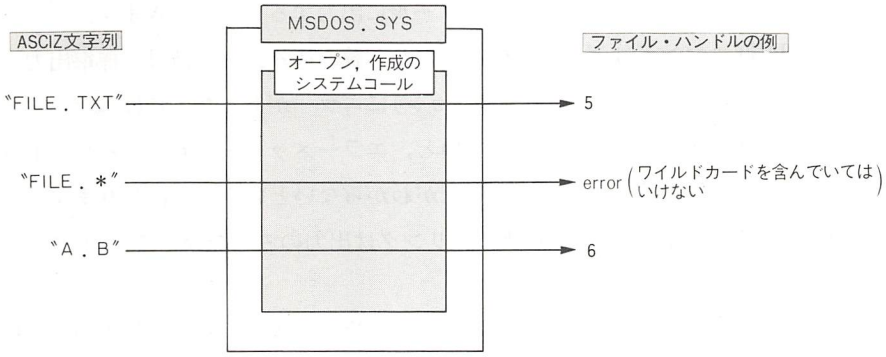
ファイル・ハンドルとは，実は単に0からはじまる2バイトの整数のことであり，BASICのファイル番号のようなもののことです．このファイル・ハンドルは，ファイルをオープン(XENIXコンパチブルコールを用いる)または作成したときに〝MSDOS．SYS〟の方から与えられ，それ以後のファイル・アクセスは，このファイル・ハンドルを用いて行われます．たとえば，ファイル・ハンドル7番のファイルから10バイトデータを読む，というようにです．いったんファイルをオープンすれば，後はそのファイル・ハンドルの値(2バイト)さえ管理すればよいので，FCBを用いたファイルのアクセスに比べてその手間はぐんと減ります．

ファイルをオープンするには，FCBと違って単にオープンしたいファイルのASCIZ文字列を使います．たとえば，ファイル〝¥USR¥A．C〟をASCIZ文字列で表すと次のようになりますが，このような文字列の先頭アドレスをオープン・システムコール(ファンクション3DH)に渡すと，〝MSDOS．SYS〟は，該当するファイルを捜し出してきてこのファイル用のファイル・ハン

ドルを返します．

[5C　55　53　52　5C　41　2E　43　00] (16進)

また，ユーザープログラムではこのファイル・ハンドルを格納しておき，以後のファイル・アクセスはこれを用いて行います(Fig. 3.29)．



※ ファイル名をASCIZ文字列で表すのは，MS-DOSのもととなっているUNIXの特徴である．そのUNIXを記述しているC言語では，文字列はASCIZ文字列で管理され，BASICのように文字列長＋文字列本体というようには管理されていない．
C言語での文字列は"ABC"と書くと〔41H，42H，43H，00H〕とASCIZ文字列となるため，図中の"A．B"などはC言語で書いたASCIZ文字列と同じであると了解してほしい．なお，システムコールを行うときは，このASCIZ文字列の先頭のアドレスをパラメータとして渡す．

Fig. 3．29　ASCIZ文字列とファイル・ハンドル

ファイル・ハンドルの値は0～19のうちのどれかですが，どれになるかはシステム側で決められます．しかし，通常プロセスが起動されたときにシステム側でTable 3.3に示す0～4はすでにオープンされているので，これらのハンドルに割り当てられているファイルはすぐにアクセスすることができます．

Table　3．3　標準ファイル・ハンドル

番号	ハンドル	機　能
0	標準入力	通常CONになっているが，リダイレクトにより，ファイルにすることも可能．プログラムに対するデフォルトの入力を決める
1	標準出力	通常CONだが，ファイルに切り換えることも可能．プログラムの出力する文字(列)をどこへ出力するかを決める
2	標準エラー出力	つねにCONになっていて，"COMMAND.COM"ではリダイレクトできない．このため，標準出力がファイルになっていてもこの標準エラー出力に文字列を出力すると，その文字列がスクリーンに出力される．エラーメッセージの出力に向いている
3	標準補助装置	RS-232Cのようなデバイスを指す．システムによってはこのデバイスはオプションとなっている．デバイス名はAUX．リダイレクトは機械語レベルで可能
4	標準プリンタ	プリンタへの文字列の出力用の割り当て番号．デバイス名PRNに相当する．機械語を使うとリダイレクトができる

標準入力は，通常コンソールになっていますが(つまり，標準入力からデータを読み込むとコンソールから入力が行われる)，入力のリダイレクトを行うとコンソールではなくファイルから入力されるようになります．標準出力は通常コンソールですが，出力のリダイレクトを行うとファイルへデータが書き込まれます．標準エラー出力はつねにコンソールになっており，I／Oのリダイレクトが行われてもエラーメッセージをコンソールに出力することができます．

たとえば，FINDコマンドはデータを標準入力から読み込み，実行結果を標準出力へ送りますが，パラメータ(見つけるべき文字列やオプション)にエラーがあった場合はエラーメッセージを標準エラー出力に送ります．こうしておかないと，エラーメッセージもリダイレクトされてファイルにはいってしまい，エラーが起こったことがわからないということになります．なお標準補助装置はAUXのことを指します．また標準プリンタは出力のみ可能なので，つねにプリンタにデータを送るようになっています．

ファイル・ハンドルを用いると，すべてのデバイスやファイルに対しての入出力を統一的に扱うことができます．つまり，同じシステムコールを用いながらファイル・ハンドルの値を変えるだけでいろいろなデバイスやファイルのアクセスができるのです．

標準ファイル・ハンドルが0～4を占めているため，ユーザーが新たにオープンしたファイルのファイル・ハンドルは通常5からはじまります．しかし，親プロセスでいくつかのファイルがオープンされていて子プロセスがそれらのファイルを受け継ぐ場合，次のファイル・ハンドルは5ではなくもっと大きな数字からはじまります(Fig. 3.30)．

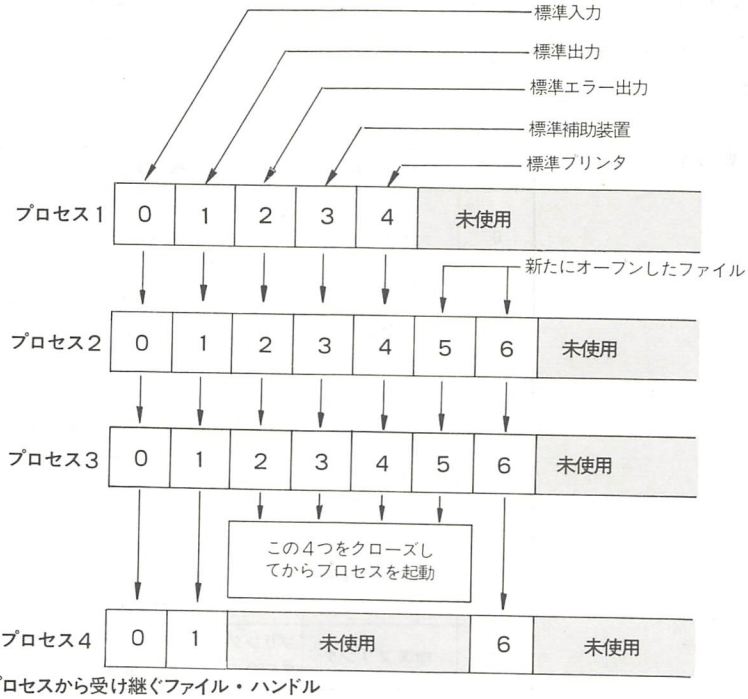


Fig. 3.30　親プロセスから受け継ぐファイル・ハンドル

3.7　システムコール

MS-DOS のシステムコールは，ver. 1.25 のときからあるものと ver. 2.0 で新しく導入された XENIX コンパチブルなものの 2 種類が融合してできているため，各ファンクションの名前を見るとかなり重複しているように思えますが，それらは機能は似ていてもそれら 2 種類を混用することはできないため，どちらか一方を使うことをお勧めします．使いやすさや将来性からいえば XENIX コンパチブルのシステムコールを用いる方がよいのですが，CP／M や ver. 1.25 に慣れたユーザーやソフトウェアの蓄積があるユーザーにとっては，ver. 1.25 のシステムコールの方がよいでしょう．しかし，ver. 1.25 のシステムコールでは実現できない機能(たとえば，新しいディレクトリの作成や削除，カレント・ディレクトリの変更，メモリ管理，プロセス管理など)があるので，なるべく ver. 2.0 の新しいシステムコールを使うようにした方がよいでしょう．システムコール一覧を Table 3.4 に示します．

Table 3．4　システムコール一覧

番　号	ファンクション	
00H	プログラムの終了	Terminate Program
01H	キーボード入力とコンソールへのエコー	Read Keyboard and Echo
02H	文字の出力	Display Character
03H	補助入力	Auxiliary Input
04H	補助出力	Auxiliary Output
05H	プリンタ出力	Print Character
06H	コンソールとの直接入出力	Direct Console I/O
07H	コンソールからの直接入力	Direct Console Input.
08H	エコーなしのキーボード入力	Read Keyboard
09H	文字列の出力	Display String
0AH	バッファード・キーボード入力	Buffered Keyboard Input
0BH	キーボードのステータス・チェック	Check Keyboard Status
0CH	バッファを空にしてキーボード入力	Flush buffer, Read Keyboard
0DH	ディスクのリセット	Disk Reset
0EH	ドライブの選択	Select Disk
0FH	ファイルのオープン	Open File
10H	ファイルのクローズ	Close File
11H	最初のディレクトリ・エントリの検索	Search for First Entry
12H	次のディレクトリ・エントリの検索	Search for Next Entry
13H	ファイルの削除	Delete File
14H	シーケンシャルな読出し	Sequential Read
15H	シーケンシャルな書込み	Sequential Write
16H	ファイルの作成	Create File
17H	ファイル名の変更	Rename File
19H	カレント・ドライブ番号の読出し	Current Disk
1AH	ディスク転送アドレスの設定	Set Disk Transfer Address
21H	ランダムな読出し	Random Read

番　号	ファンクション	
22H	ランダムな書込み	Random Write
23H	ファイルサイズの計算	File Size
24H	相対レコード・フィールドの設定	Set Relative Record
25H	割込みベクトルの設定	Set Interrupt Vector
27H	ランダムなブロック読出し	Random Block Read
28H	ランダムなブロック書込み	Random Block Write
29H	ファイル名の解析	Parse File Name
2AH	日付の読出し	Get Date
2BH	日付の設定	Set Date
2CH	時刻の読出し	Get Time
2DH	時刻の設定	Set Time
2EH	ベリファイフラグの設定	Set/Reset Verify Flag
2FH	ディスク転送アドレスの読出し	Get Disk Transfer Address
30H	DOSのバージョン番号の読出し	Get DOS Version Number
31H	プログラムの常駐終了	Keep Process
33H	ブレークチェックの制御	CONTROL-C check
35H	割込みベクトルの読出し	Get Interrupt Vector
36H	ディスクの残り容量の読出し	Get Disk Free Space
38H	国別情報の読出し	Return Country-Dependent Information
39H	サブ・ディレクトリの作成	Create Sub-Directory
3AH	サブ・ディレクトリの削除	Remove Directory Entry
3BH	カレント・ディレクトリの変更	Change the Current Directory
3CH	ファイルの作成	Creat a File
3DH	ファイルのオープン	Open a File
3EH	ファイルのクローズ	Close a File Handle
3FH	ファイル／デバイスの読出し	Read From File/Device
40H	ファイル／デバイスの書込み	Write to a File/Device
41H	ファイルの削除	Delete directory Entry
42H	ファイル・ポインタの移動	Move a File Pointer
43H	ファイルの属性の変更	Change Attributes
44H	デバイスに対するI／Oコントロール	I/O Control for Devices
45H	ファイル・ハンドルのコピー	Duplicate a File Handle
46H	指定したファイル・ハンドルへのコピー	Force a Duplicate of a File Handle
47H	カレント・ディレクトリの読出し	Return Text of Current Directory
48H	メモリブロックの割当て	Allocate Memory
49H	メモリブロックの解放	Free Allocated Memory
4AH	メモリブロックのサイズの変更	Modify Allocated Memory Blocks
4BH	プログラムのロードと実行	Load and Execute a Program
4CH	プロセスの終了	Terminate a Process
4DH	子プロセスの終了コードの読出し	Retrieve the Return Code of a Child
4EH	一致するファイルの検索	Find Match File
4FH	次に一致するファイルの検索	Step Through a Directory Matching Files
54H	ベリファイフラグの読出し	Return Current Setting of Verify Flag
56H	ディレクトリ・エントリの移動	Move a Directory Entry
57H	ファイルの時間情報の読出し／設定	Get/Set Date/Time of File

これらはその機能からいくつかに分けることができますが，それについては3.7.2以降で順次取り上げていきます．

3.7.1　システムコールの方法

システムコールを使うには，AH レジスタにファンクション番号をセットし，他のレジスタにパラメータをセットした後，内部割込み INT　21H を行います．INT　21H 命令を使わなくとも，PSP のオフセット 50H を FAR CALL するという方法もありますが，あまりメリットはありません．またもう 1 つ，PSP のオフセット 05H をコールするという方法もありますが，使えるシステムコールに制限があります．詳しくは 7 章を参照してください．

通常，ディスクやファイル関係のシステムコールを用いると，期待した動作が正しく行われたか否かを示すエラーステータスが返ってきます．ver. 1.25 のシステムコールでは AL＝00H のときが正常終了，AL＝FFH のときは何らかのエラーが発生したというように，AL レジスタにエラーステータスが返ります．

ver. 2.0 のシステムコールでは，キャリーフラグが 0（キャリーがリセットされている）のときが正常終了，キャリーフラグが 1 のとき何らかのエラーが発生したことを示します．なお，エラーが発生した場合，キャリーフラグがセットされるだけではなく，AX レジスタにエラーの発生した原因を示すエラーコードが入れられます．

3.7.2　コンソール入出力

AH ＝ 01H，02H，06H～0CH

これらのシステムコールは，コンソール（デバイス名は〝CON〟）との入出力を行うためのものです．

1 文字入力については，エコーバックする（入力された文字をスクリーンに出力する），しない，CTRL-C が有効，無効（CTRL-C が入力できる）の組合せで計 3 通り，さらにファンクション 0CH を使えば，タイプアヘッドバッファを空にしてから今の 3 つの動作を行うなど，多くの機能があります（Table 3.5）．

Table 3.5　1 文字入力

エコー ＼ CTRL-C	有　効	無　効
す　る	1	——
しない	8	6,※ 7

※キーが押されていなければ
すぐシステムコールから戻る

ファンクション 02H と 06H のシステムコールは 1 文字出力ですが，CTRL-C が有効か無効かの違いがあります．

ファンクション 0BH は，キーボードバッファに文字があるかないかを調べます．

文字列を出力するには，ファンクション 09H を用います．文字列の終わりの印は〝$〟ですので，〝$〟を出力したい場合はファンクション 02H を使用します．

ファンクション0AHは，コンソールから最大255文字までの文字列を入力するためのもので，入力の終わりにはCTRL-M(CR)を入力します．なお，入力時にはテンプレート機能が使えます．またこのバッファの内容を変更しないでおくと，次のシステムコールの際に今回入力したデータを変更して(テンプレートを使って)入力することができます．このシステムコールを使うために必要となるバッファのフォーマットをFig. 3.31に示します．

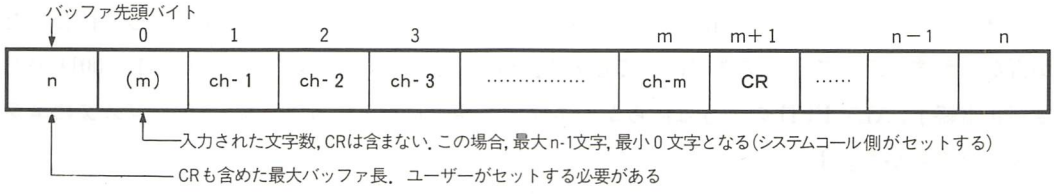


Fig. 3.31　入力用バッファ

以上がコンソール入出力のシステムコールですが，特に断っていない場合，入出力中にCTRL-Cが入力されると，制御はCTRL-C処理ルーチン(INT　23H)へ移り，BREAKスイッチがONのときは，つねにCTRL-Cによって内部割込み23Hが実行されます(INT　23Hの処理ルーチンへ制御が移る)．

このコンソール入出力のシステムコールは，ver. 1.25のMS-DOSでは，実際にコンソールとデータをやり取りしていましたが，ver. 2.0ではつねにコンソールとの入出力が行われているわけではありません．入出力のリダイレクトが行われると，データはコンソールからではなく，ファイルから読み込まれたり，出力がスクリーンではなく，ファイルに書き込まれたりします．この入出力の変更は"MSDOS．SYS"側で自動的に行われ，プログラム側で意識する必要はありません(リダイレクトされているかどうかプログラム側からはわからない)(Fig. 3.32)．

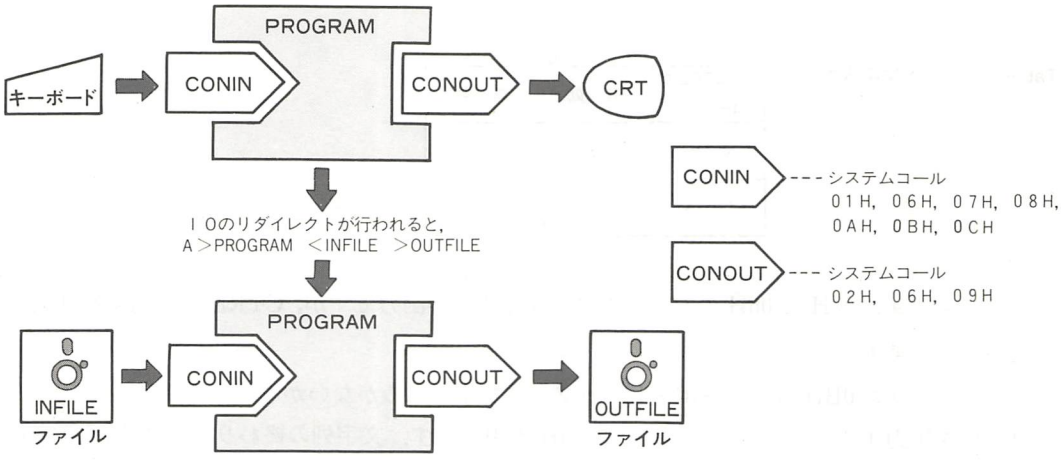


Fig. 3.32　コンソール入出力とリダイレクト

入力がリダイレクトされると，キーボードのステータスチェックにおいて，少し変更が起こります．しかし，これらはリダイレクト時でもプログラムが正しく動くようにするためであり，何ら気にすることはありません．

入力がファイルにリダイレクトされている場合，キーボードバッファのステータスチェックでは，つねに〝データが用意されている〟というステータスが返され，キーボードのバッファフラッシュのシステムコールでは何も起こりません．

出力がリダイレクトによりファイルになっている場合，コンソール出力もファイルに送られるため，エラーメッセージを画面に出力しようと思ってもうまくいきません．ですからエラーメッセージは，ファイル・ハンドルの番号2のファイルへ書き込まなければなりません．

3.7.3　デバイスとの入出力

AH = 03H, 04H, 05H

これらのコールの目的は，補助装置(AUX)との入出力と，プリンタ(PRN)への出力です．

補助装置が何であるかは機種により異なりますが，多くの場合はRS-232Cとなっています．RS-232C を用いる場合，使用前にデータ幅，ボーレート，ストップビット長，回線モード(全二重か半二重か)，XON／XOFF の制御など，種々のモードを決めておく必要がありますが，それらの設定については，MS-DOS では明確な定義はなく，メーカーに依存しています．たとえば，NEC の PC-9800 シリーズでは SPEED コマンド(NEC 製)により，回線仕様を決定しますが，このようなコマンドがなく，機械語か BASIC で設定するという機種もあります．

なお，ver. 2.0 では標準ファイル・ハンドルの中に標準補助入出力と標準プリンタ出力があるため，これらのシステムコールは使わなくても済むようになっています．

3.7.4　FCB によるファイル・アクセス

AH = 0DH～24H, 27H～29H, 2FH

FCB を使ったファイルのアクセスは，MS-DOS ver. 1.25 や CP／M との互換性のためにあります．ですから，互換性の必要のないユーザーは XENIX コンパチブルなシステムコールの利用をお勧めします．

FCB では階層構造のディレクトリに対応できないと前に述べましたが，実際には FCB を用いてもカレント・ディレクトリ中のファイルのみは扱うことができます(オープンやクローズ，読出し，書込み，作成，検索などが可能)が，カレント・ディレクトリでもディレクトリ名(やボリューム名)は参照できません．

ファイルのアクセスに先立って，まず FCB にファイル名をセットしなければなりませんが，ファンクション 29H のファイル名の解析をコールすれば，それが簡単に行えます．

いったん FCB にファイル名を登録すれば，ファイルの検索(最初はファンクション 11H, 2 番

目のファイルからはファンクション12Hを使う)やファイルの削除(ファンクション13H)などが利用できます．ファイル名を変更したい場合は，FCBのオフセット11Hから新しい名前をセットしておき(ファンクション29H)，ファンクション17Hをコールすれば実現できます．このときファイル名にワイルドカードを用いると，一度に複数のファイル名の変更ができます．たとえばファイル名が〝T．*〟のファイルを〝TEST．*〟に名前変更させると，〝T．ASM〟や〝T．OBJ〟などがすべて〝TEST．ASM〟，〝TEST．OBJ〟という名前に変更されます．

FCBにセットされたファイル名がワイルドカードを含まない場合，そのFCBのファイルをオープンして(ファンクション0FH)読出しや書込みの対象とすることができますが，ファイルの作成(ファンクション16H)を用いてもオープンされたFCBを作ることができます．ファイルを作成する場合，すでに目的のファイルが存在していれば，そのファイル内容はすべて破棄され，ファイルサイズが0となります．

FCBがオープンされればそのファイルを読み出したり書き込んだりすることができますが，その前にデータを転送するアドレス(DTA：Disk Transfer Address)をセットしなければなりません．Fig. 3.33に示すようにMS-DOSでは読出しのときは読出しレコードがDTAに転送され，書込みのときは逆にDTA上のレコードがファイルのレコード・ポインタの位置へ書き込まれます．DTAをセットするシステムコール(ファンクション1AH)ではDTAのスタート・アドレス(セグメント：オフセットの形式)を指定しますが，大きさ(DTAのサイズ)は特に指定しないので，レコードのサイズとの関係には注意してください．

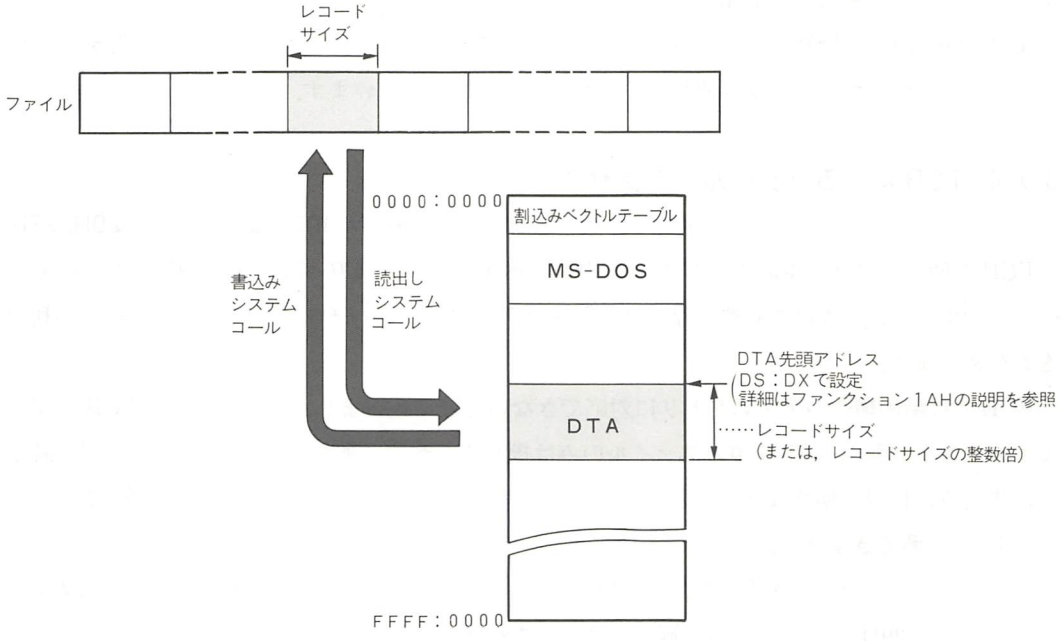


Fig. 3.33　DTAとファイルの読出し／書込み

MS-DOSでは，デフォルトのDTAとして(ファンクション1AHを用いない場合)PSPのオフセット80Hからはじまる128バイト(PSPの後半全部)が設定されていますから，レコードサイズがデフォルト(128バイト)と同じか，あるいはそれ以下であれば，このDTAをそのまま使用することができます．もしレコードサイズが129バイト以上である場合は，プログラム中で新たなDTAを用意しなければなりません．

DTAの設定およびFCBのオープンが済んだら，いよいよファイルのアクセスですが，そのために以下の6つのファンクションが用意されています．

- シーケンシャルな読出し(ファンクション14H)／書込み(ファンクション15H)
  ファイルをシーケンシャルにアクセスします．システムコールを行う前に，FCB中のレコードサイズ(デフォルトは128バイト)とカレントレコード(デフォルトなし)の各フィールドをセットしておかなければなりません．
- ランダムな読出し(ファンクション21H)／書込み(ファンクション22H)
  ファイル中の任意の1レコードをアクセスします．やはり，システムコールを行う前にFCBのレコードサイズと相対レコード(デフォルトなし)の両フィールドを正しい値にセットしておかなければなりません．
- ランダムなブロック読出し(ファンクション27H)／書込み(ファンクション28H)
  ランダムな読出し／書込みで一度に何レコードでもアクセスできるようにしたものであり，CXレジスタにアクセスするレコード数をセットします．なお，CX＝0で書き込もうとすると，ファイルが現在のレコード・ポインタの位置で切り捨てられます(または延長される)．

ファイルへのアクセスが終わったならば，そのファイルをクローズしなければなりません．システムコールのファイルクローズ(ファンクション10H)によりファイルをクローズすることができます．このシステムコールにより，書込みのあったファイルのディレクトリが更新され(ファイルサイズや最終変更日付／時間など)，ディスク上へ書き込まれます．また，FATやファイルのデータの一部(ダーティーバッファのこと．詳細は後述)も必要ならばディスクに書き込まれます．

ファイルがクローズされるとFCBはオープンされていない状態になり，そのFCBは他の目的のために使用することが可能となります．

ファンクション0DHのディスクリセットは，〝MSDOS．SYS〟内のすべてのダーティバッファ(dirty buffer)をディスクへ書き込むために使用します．詳細は3.8を参照してください，なお通常このシステムコールは，CTRL-C処理ルーチンで使用されています．

ディスクの選択(ファンクション0EH)／カレントディスク(ファンクション19H)は，カレント・ドライブの設定，参照のために使われます．

ファンクション2FHのシステムコールは，DTAの参照のために用いられます．以下の一連の手続き((1)〜(4))により，各プログラムモジュールごとに独立したDTAを確保することができます．

(1) DTAを一時退避させる
(2)新しいDTAをセットする
(3)ファイル・アクセスを行う
(4) DTAをもとに戻す

相対レコードのセット(ファンクション24H)は，カレントブロックとカレントレコードの組から決まるレコード・ポインタと同じレコードを指すように相対レコードをセットするものです．これは，シーケンシャル・アクセスをしていたファイルの途中から，ランダムブロック・アクセス(一度に2レコード以上アクセスできる)にアクセス方法を変えてアクセス速度を上げたいというようなときに便利です．

3.7.5　時間／フラグに関する制御，その他

AH = 25H, 2AH〜2EH, 30H, 32H〜38H, 54H

2AH〜2DHのファンクションは，日付と時間の設定／参照のために用いられます．

2EH, 54Hのファンクションは，ディスクのベリファイフラグの設定／参照に用いるものですが，ベリファイフラグがON(1)のときリードアフターライト(read after write)が行われ，データがディスクに正しく書き込まれたか否かのチェックが行われます．

33Hのファンクションで，CTRL-Cによる割込みをチェックするモードのON／OFFを行います．OFFのときはコンソール関係の入出力システムコールでのみブレークチェックが行われますが，ONになっているとすべてのシステムコールの実行前に(つまり，INT　21Hで"MSDOS．SYS"へ制御が移ってすぐ)ブレークチェックが行われ，CTRL-Cが押されていればINT　23H処理ルーチンに制御が移ります．

25Hと35Hのファンクションは，8086CPUの割込みルーチン(256種)のアドレスの設定／参照を行います．これはシステムコールを用いなくても簡単に行うことができますが，DOSとユーザー領域は分離しているべきであると考えられるので，システムコールによらないベクトルテーブルの参照，更新はあまり感心できません．

ファンクション30Hは，現在プログラムが動いているMS-DOSのバージョン番号を返すもので，プログラムを暴走させないためにも必要な機能です(ver. 1.25下でver. 2.0のプログラムはほとんど動かない)．MS-DOS ver. 2.0に付属しているコマンドは，たいていプログラムの先頭でこのバージョンチェックを行っています．

ファンクション36Hは，ディスク上に残りの容量がどのくらいあるかを調べるときに用います．

ファンクション38Hの国別情報というのは，各国別の時間や日付の表現方法，通貨記号(〝$〟や〝¥〟など)および小数点の記号や3桁ごとの区切り記号は何か，そのほか国によって表現方法の異なるものについての情報を得るためのファンクションです．このシステムコールにより得られる情報をもとに，日本とアメリカでの日付，時刻と金額の表現例をFig. 3.34に示します．

	日　本	U　S　A
日付と時刻 例：1984年4月29日 午後1時30分	84－04－29 13：30	4－29－84　日付は，month／day／yearの順 日付分離記号は〝－〟 1：30p　12時間制 時刻分離記号は〝：〟
金額の例	¥1,234	$1,234.56　3桁ごとの分離記号は〝,〟 小数点の記号は〝.〟 小数点以下2桁 通貨記号は〝$〟 このような項目に関する情報が得られる

Fig. 3.34　国別情報

3.7.6　ファイル・ハンドルによるファイル・アクセス

AH = 3CH～46H, 4EH, 4FH, 56H, 57H

すでに述べたように，ファイル・ハンドルはファイルを表すための整数のことであり，オープンファイル・システムコールから返されます．このハンドルは，たいていBXレジスタに入れてシステムコールを行います．1つのプログラムで使用できるファイルの数は20個ですが，5つはすでにシステム側で予約済みなので，1つのプログラムでオープンや作成に使用できるファイルの数は15個までです．しかし，親プロセスのオープンしたファイルを受け継ぐ場合，使えるファイルは15より少なくなりますし，〝CONFIG．SYS〟ファイルでFILESを設定しておかないと，プログラム中で使用できるファイルの数は5個になってしまいます．ただし，この数はXENIXコンパチブル・システムコールを用いた場合のものであり，FCBを用いてファイルを扱う場合には制限はありません．

3CHと3DHのファンクションは，ともにファイルをオープンするために用いられます．ファンクション3DHは既存のファイルをオープンするものですが，ファンクション3CHは新しいファイルを作成するために使われるもので，このときすでにファイルが存在する場合はその長さが0になります．FCBを用いたファイルオープンとは違い，ファイルを次の3つのうちどのモードでオープンするかを指定しなければなりません．このモードは，DOS側でファイルバッファの管理を効率的に行うために使用され，そうすることによって標準入力に書き込んだり標準出力から読

み出そうとしたりするような暴挙からファイルを守っています．

(1)読出し　　(2)書込み　　(3)読出し／書込み両用

ファンクション3CHのファイル作成システムコールではファイルは(3)のモードでオープンされているので，データを書き込んだ後ファイル・ポインタを前の方へ戻せばそのファイルの内容を読み出すことができます．

なおMS-DOSでは，デバイスもファイルと同じようにこのオープン・システムコールでオープンしてデータを読み書きすることができます．

ファイルは，一度オープンされると3FH, 40Hのファンクションを用いてアクセスすることができます．このとき，データを読み書きする領域(DTAのようなもの)と何バイト読み書きするかをシステムコールを行うごとに指定しますが，ファイルのどこからアクセスを行うかは，前回のシステムコールでアクセスした次のバイトからということになります．つまり，基本的にはシーケンシャル・アクセスが行われます．

ファンクション42Hのファイル・ポインタの移動は，ファイルをランダム・アクセスする場合に有効です．通常ファイル・ポインタは，オープン時にファイルの先頭(オフセット0)にセットされ，読出し／書込みが行われるとアクセスされたバイト数だけファイル・ポインタが進められますが，このシステムコールを行うと，ファイルの任意の位置からアクセスをはじめることができます．ファイルサイズは最大1Gバイトにもなり得るため，ファイル・ポインタの位置は30ビットの値を使用して表されます．このシステムコールでは，新しいファイル・ポインタの位置を，(1)ファイルの先頭から，(2)現在ファイル・ポインタの位置から，(3)ファイルの終わりから，それぞれ4バイトのオフセットを足した位置へ移動させることができるため，ランダム・アクセスやファイルの最後の2Kバイトをダンプしたりということが簡単に行えます．

ファンクション3EHの機能はファイル・ハンドルのクローズであり，これによりファイルの最終変更日付やサイズ，FATなどがディスクに書き込まれ，使用していたファイル・ハンドルの値が次のオープンのために開放されます．

ファンクション44Hのデバイスに対するI／Oコントロールとは，Fig. 3.35に示すように主にCONやPRN, AUXなどのデバイスをコントロールしているプログラム("IO．SYS"内のデバイス・ドライバ)とデータ(デバイス情報)のやりとりを行うためのものです．混同しないでいただきたいのは，データはデバイス自身(コンソールやプリンタ，RS-232C回線など)ではなく，そのデバイスをコントロールしているデバイス・ドライバとやりとりされるということです．単にデバイスとデータをやりとりするためにはすでにシステムコールが用意されており，ユーザーが増設したデバイスについてはファイルの形式を採ることでデータが読み書きできます．

使用例としては，RS-232Cドライバルーチンにボーレート，ビット長，ストップビットの有無などのRS-232Cの回線仕様を渡すというようなものが考えられます．しかし，MS-DOS ver. 2.0

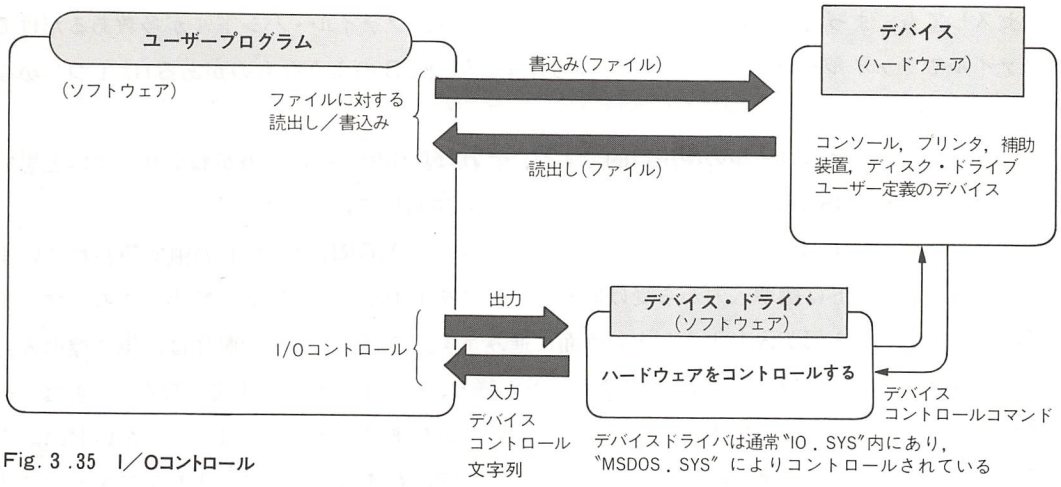


Fig. 3 .35　I／Oコントロール

が標準で備えているデバイス・ドライバはこのI／Oコントロールの機能が使えないので(サポートされていない)，このような目的に使うためにはユーザー自身で新しいデバイス・ドライバを作る必要があります．

またその他の機能として，ファイル・ハンドルを渡してそれがコンソールか否かを調べることができるので，現在標準入力がコンソールであるかディスク上のファイルであるかをプログラムの中から調べることができます．通常，I／Oのリダイレクトは親プロセスが行うため，子プロセスでは標準入出力が何であるかはわかりませんが，このI／Oコントロールの機能を使用すれば，それがコンソールであるかファイルであるかの区別はつきます．

また，コンソールのモードをRAWにするかCOOKEDにするかもこのシステムコールで行えます．通常コンソールはCOOKEDモードになっており，このモードの下ではコンソールへの入出力はつねにCTRL-Sによる一時中断が行えますし，CTRL-Cによる割込みのチェックも1文字ごとに行われています．また，コンソール入力もバッファリングされており，ファンクション3FHのファイル読出しシステムコールでデバイスCONを読む場合，1行のすべてが入力されるまでバックスペースなどのコントロール・キャラクタも使えますし，CTRL-C割込みも使えます．

コンソールのモードをCOOKEDからRAWに変更すると，ファンクション40Hによる書込みの途中ではCTRL-SやCTRL-Cによる中断ができなくなり，ファンクション3FHによる読出しではいっさいのバッファリング動作が行われなくなりますが，これらのチェックが行われない分だけコンソールとの入出力動作が速くなります．

45Hと46Hのファンクションは，ともにあるファイル・ハンドルのコピーを行うものです．ファイル・ハンドルのコピーとは，あるファイルを操作するために2つ以上のファイル・ハンドルを用意することであり，ファイル・ハンドルの値が違うだけでほかはまったく対等に扱います．それらのコピーされたハンドルともとのハンドルは同一のファイルを指しており，2つのハンドルで交互にアクセスしても1つのハンドルでアクセスしても，さらに多数のハンドルで順番にア

クセスしても，まったく同じ結果が得られます．つまり，ファイル・ハンドルが多数あるだけでファイルやファイルをコントロールしている作業領域(FCBのようなものがある)は1つしかないのです．

さて，このシステムコールの使用目的ですが，これは具体例を挙げた方がわかりやすいと思いますので，MS-DOS ver. 2.0での使用例を示しながら説明していきます．

ファンクション45Hのファイル・ハンドルのコピーは，MOREコマンドの中で使われています．MOREコマンドは標準入力から受け取ったデータを1画面ごとに区切って出力するコマンドで，キーボードから何か入力すると次の画面に進みます．このコマンドの動作は，単に標準入力から20数行分のデータを読み込んでスクリーンへ送り，キーボードから1文字読んで，またデータを読んでいるというものですが，ここで1つ問題があります．というのは，いったいどのようにしてキーボードから文字を読むかということです．前にも述べたように，入力がファイルにリダイレクトされているときはキーボードからの入力はコンソールのシステムコール(01H, 06H, 08H, 0AHなど)では読むことができません．

それを避けるためにMOREコマンドでは，標準入力のファイル・ハンドル(0)を別のファイル・ハンドルにコピーしておいて，0番のファイルをクローズしてしまいます．すると，コンソール入力のシステムコールはキーボードからデータを拾うようになりますから，そこでさきほど

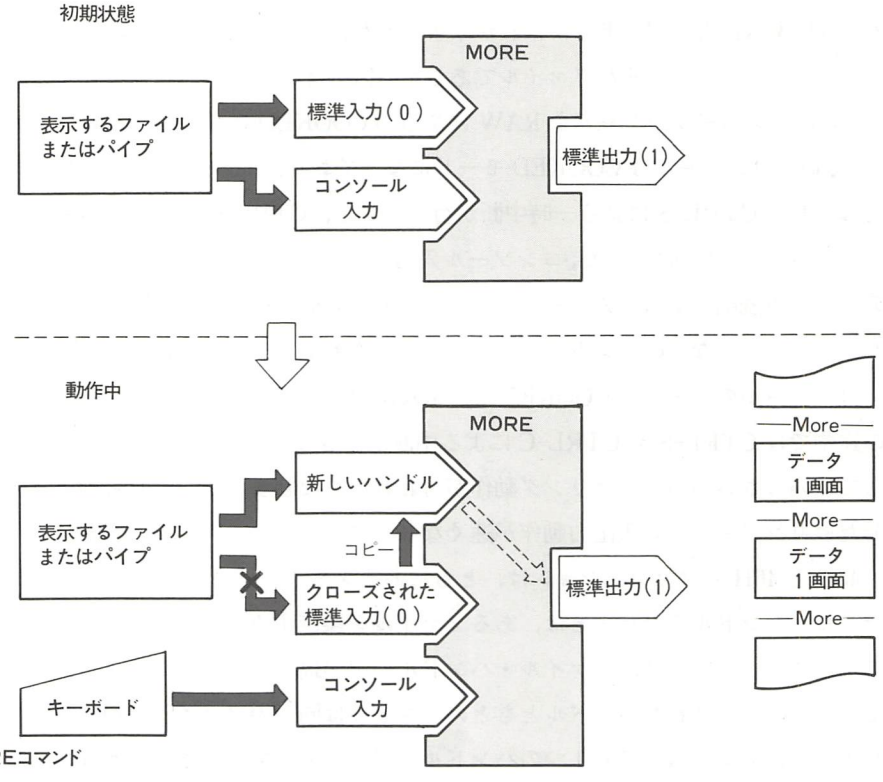


Fig. 3.36　MOREコマンド

コピーしておいたファイル・ハンドルのファイルから1画面分のデータを読み，キーボードのステータスチェックのシステムコールを用いてユーザーの入力を待つという動作をします(Fig. 3. 36)．

ファンクション46Hのファイル・ハンドルのコピーの強制は，あるファイル・ハンドルと同じものを指すファイル・ハンドルを，指定したハンドル番号にコピーします．ファンクション45Hと異なるのは，ファイル・ハンドルのコピー先のハンドルの番号を指定する点です．ファンクション45Hではシステム側で使っていないファイル・ハンドルにコピーしますが，ファンクション46Hにおいては，もしコピー先のハンドルが使用中ならばそのファイル・ハンドルのファイルがクローズされた後でコピーを行います．

〝COMMAND．COM〞では，ユーザーのコマンドを実行する前にI／Oのリダイレクトが行われているかを調べ，必要があればファイルをオープンしてファンクション46Hを使って0と1のファイル・ハンドルへ今オープンしたファイルを割り付けています．その後プログラムをロードして実行することにより，そのプログラムのI／Oをリダイレクトすることができます．

以上のシステムコールはファイル・ハンドルを用いたものでしたが，以下のものはファイルをオープンせず，単にASCIZ文字列を用いるシステムコールです．

ファンクション41Hのファイルの削除は，パス名を与えてそのファイルを削除します．なお，パス名にはワイルドカードは許されません．

ファンクション43Hは，ファイルの属性の設定／参照を行います．これも与えるパス名にワイルドカードは使えません．ファイルの属性としてはTable 3.6のようなものがあります．

Table 3．6　ファイルの属性

属　性	意　味
01H	読出し専用ファイル
02H	不可視属性．通常のディレクトリサーチから除外される
04H	システムファイル
08H	ボリュームラベル
10H	ディレクトリ
20H	保存ビット

このうちファンクション43Hでは，ビット0(read only)，ビット1(hidden)，ビット2(system)，ビット5(archive)のみが変更できます．

4EHと4FHのファンクションでは，ワイルドカードを含むパス名のASCIZ文字列を与えて該当するファイルを捜し，ファイルの属性や最終変更時刻，ファイルサイズなどの情報を得ることができます．ファンクション4EHと4FHの使用法は，まず4EHのシステムコールで最初のファイルを捜し，2番目以降は4FHのシステムコールを用います．

3.7.7　ディレクトリ管理

AH = 39H～3BH, 47H

ファンクション39H～3BHはそれぞれ，MS-DOSの内部コマンド〝MKDIR〟，〝RMDIR〟，〝CHDIR〟の3つのコマンドに相当し，パラメータはパス名のASCIZ文字列で渡します．

ファンクション47Hはカレント・ディレクトリのASCIZ文字列を返すシステムコールであり，パラメータなしのCHDIRコマンドに相当します．

3.7.8　プロセス管理

AH = 00H, 31H, 4BH～4DH

MS-DOSはシングルタスクのDOSであるため，プロセス管理といってもシステムコールは起動，終了，子プロセスの終了状態のチェックの3つしか用意されていません．プロセスの起動には，ファンクション4BHを用いますが，これには，Fig. 3.37のようにPSPや環境を用意してまったく新しい子プロセスを作ることもできますし，オーバーレイ形式でプログラムをメモリへロードすることもできます．

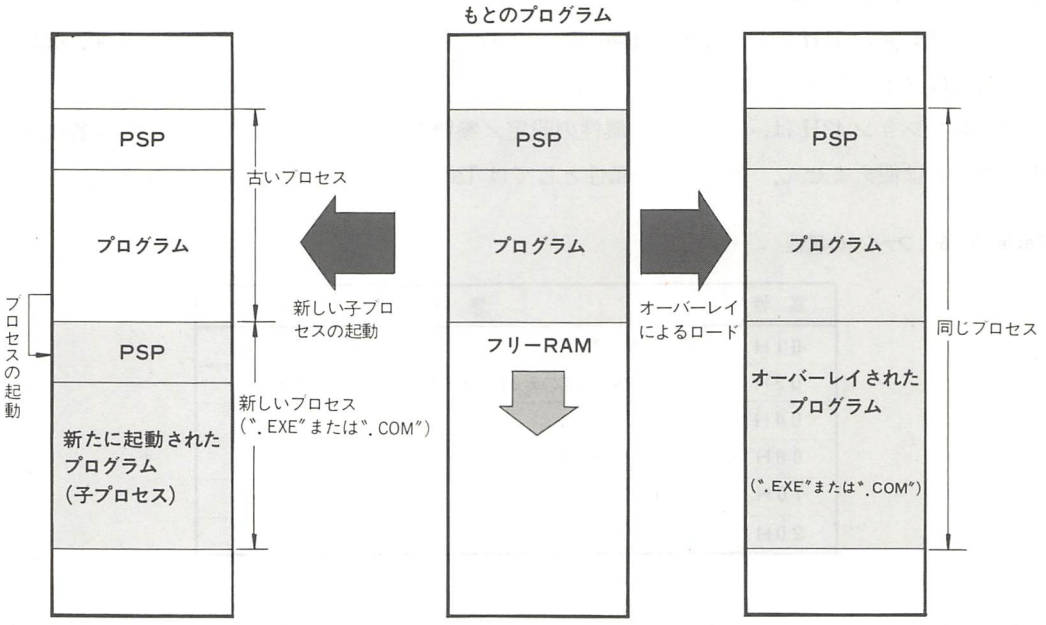


Fig. 3.37　新しい子プロセスとオーバレイ

プログラムがメモリにロードされる場合，そのプログラムのロードされる位置からフリーRAMの最後までが割り当てられています．たとえば，〝COMMAND．COM〟からDEBUGを起動し，制御がDEBUGの最初の命令に移った瞬間を考えます．このとき，システムのフリーRAMはすべてDEBUGに割り当てられており，DEBUGのPSPのオフセット02Hの1ワードには

DEBUGで使用できるRAMの上限がセットされています(Fig. 3.38①)．この状態では新たなプログラムをロードするメモリ領域がないため，ファンクション4AHのシステムコールを用いて，子プロセスを起動する前にはロードするためにいくらかのメモリを解放してやらなければなりません．さもないと，ロードするメモリ領域がないというエラーが起こります．

DEBUGが，ユーザーのデバックしたいプログラム〝BUG．COM〟をロードして実行する過程をFig. 3.38②，③に示します．

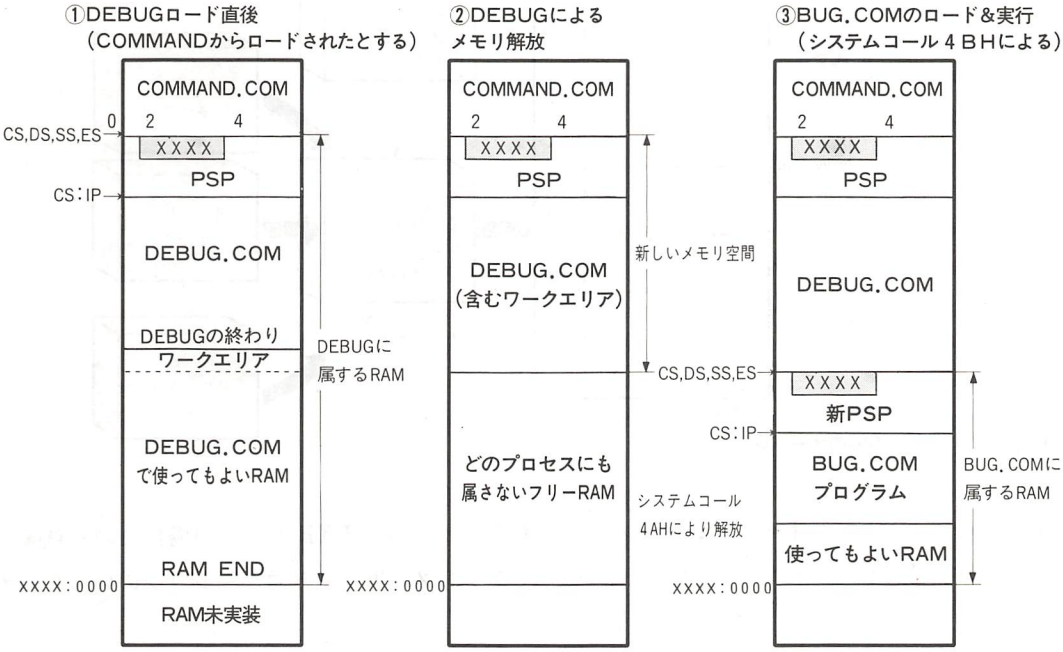


Fig. 3 .38　DEBUGと子プロセス

以上は子プロセスのロードの場合の説明でしたが，オーバレイ形式の場合はメモリの解放を行わなくても済むためもっと簡単に処理できます．この場合，単にロードするアドレスを指定してシステムコールを行うだけです．当然のことながら，PSPは作られません．

プログラムをオーバレイするには，２つのワード情報が必要です．１つはプログラムをどこにロードするかを示すセグメント・アドレス(Fig. 3.39のyyyy)ですが，もう１つはプログラムの再配置因子(relocation factor)と呼ばれるプログラムのリロケーションに関するセグメント・アドレスです．前者はプログラムをロードする物理アドレスですが，後者はそのプログラムの動作する論理アドレスのことを指し，これは通常ロードされるセグメント・アドレスと一致させておきます(Fig. 3.39のyyyyにしておく)．そうしないと，ロードされたプログラムはまったくセグメントのリロケーション操作が行われていないものとなってしまいます(つまり，セグメント・アドレス0に置かれたようにリロケーションされてしまう)．この再配置因子はセグメント・アドレスのリ

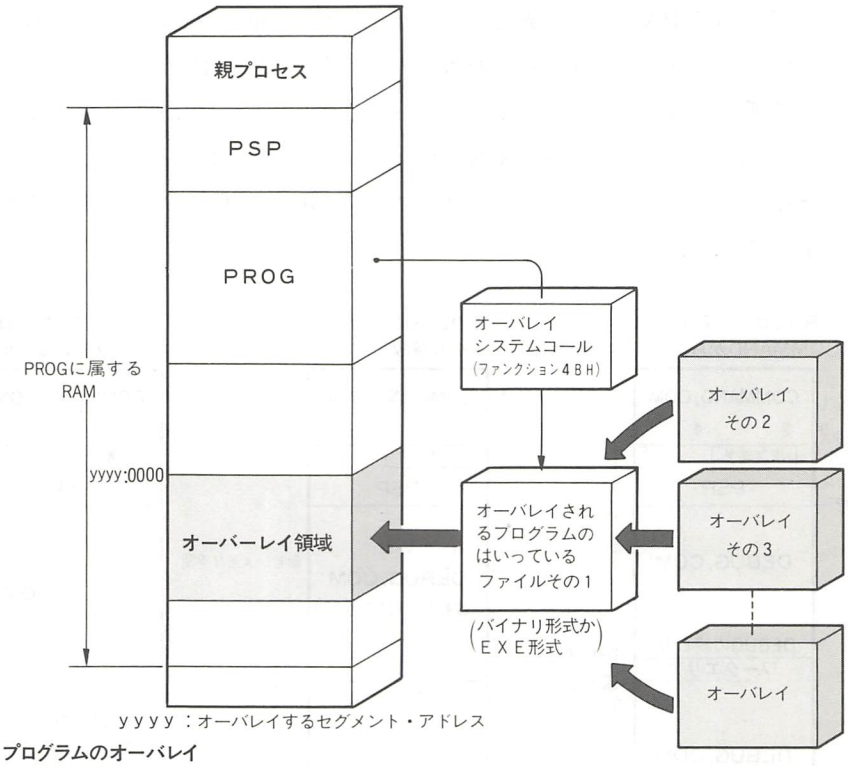


Fig. 3 .39　プログラムのオーバレイ

ロケーションに用いられるため，オーバレイされるファイルがEXEモデルの場合にのみ意味を持ちます．COMファイルのように，セグメント･アドレスに関するリロケーションのいらないオーバレイ・プログラムには意味がありません．

プロセスの実行を終了させるには，以下の6つの方法があります．

(1) PSPのオフセット0へジャンプする
(2) CSレジスタにPSPの先頭アドレスをセットして，INT　20Hを行う
(3) AH＝00Hとし，CSレジスタにPSPの先頭アドレスをセットしてINT　21Hを行う
(4) AH＝4CHとして，INT　21Hを行う
(5) AH＝00Hとして PSPのオフセット50Hへジャンプあるいはコールする
(6) AH＝4CHとして，PSPのオフセット50Hへジャンプあるいはコールする

これらのうち，CSレジスタがPSPを指さなくてもよい(4)の方法がMS-DOS ver. 2.0らしくて最もお勧めできる方法です．ほかの方法は，すべてPSPへのジャンプになるため，PSPの先頭アドレスを覚えておかなければなりません．また，ファンクション4CHのシステムコール以外の方法では，ファイルを自動的にクローズしてくれません．

どの方法でプロセスを終了しても，制御はファンクション4BHのシステムコールの次の命令

へ移ります．このときキャリーはリセットされているので，システムコールが失敗して実行できなかった場合と区別できます．

プロセスの終了方法には，実はもう1つあります．ファンクション31Hのシステムコールがそれですが，これを使うと，単にプロセスが終了するだけでなくプログラムをメモリ中に残しておくことができます．つまりINT　27Hと同じ働きをするわけです．INT　27Hは64Kを超えるプログラムをメモリに残しておくことはできませんが，こちらの方はどんなプログラムサイズにでも対応することができます．

プロセスの終了にファンクション31Hや4CHのシステムコールを用いた場合，親プロセスにリターンコードと呼ばれる1バイトの整数を返すことができます．このリターンコードを調べるファンクションが4DHなのですが，機械語を使わなくても"COMMAND．COM"のIFコマンドでその値を調べることができます．またこのシステムコールでは，リターンコードとは別に子プロセスがどういう状態で終了したのかがわかります．たとえば，正常に終了したか否かプログラムを残したまま終了したのかどうか，CTRL-Cの割込み処理ルーチンでAbort(中止)を選択したのかどうかなどです．

MS-DOSのコマンドでは，リターンコードはどれも0しか返しませんが(つまり，まったくサポートされていない)，これから自分で作るプログラムではリターンコードを返すようにすることをお勧めします．たとえば0ならば正常，1ならば不正なデータである，2ならば致命的エラーにより処理ができなかった……などと決めておけばよいでしょう．

3.7.9　メモリ管理

AH = 48H, 49H, 4AH

メモリ管理システムコールは，MS-DOSで扱うことのできるRAM領域を割当てあるいは解放することによって，プロセス間でRAMの使用領域の競合が起こらないように管理するためのものです．

複数のプログラムが1つのメモリの下に置かれるようなDOSの場合，プログラム，データ，スタックのどこか1バイトでも勝手に書き換えられてしまうと結果は保証できなくなり，ひどいときにはCPUが暴走してしまいます．この問題は，将来MS-DOSがマルチタスクになったときいっそう深刻なものとなるであろうことは容易に予想されます．また，メモリが少ない機種においては，メモリを効率よく管理できないとメモリ資源のむだづかいが起こります．

もともと8086CPUはメモリ管理をサポートするような機能を持っていませんが，プログラムはパラグラフ(16バイト)単位では割と容易にリロケートできるため，MS-DOSではこの方式でメモリを管理しています．つまり，メモリの割当ての最小単位は1パラグラフ(16バイト)になります．

ファンクション48Hのメモリブロックの割当て(メモリ・アロケーション)は，どのプロセスに

も属さないフリーRAMから，要求したパラグラフ数のメモリをプロセスに与えるためのものです．フリーRAMに空き領域がない場合にはエラーとなりますが，そうでない場合は割り当てられたメモリブロックの先頭アドレスが返ってきます．先頭アドレスがどこになるかはそのときのメモリの割当て状況により異なる可能性がありますし，先頭アドレスを指定することもできません(Fig. 3.40)．

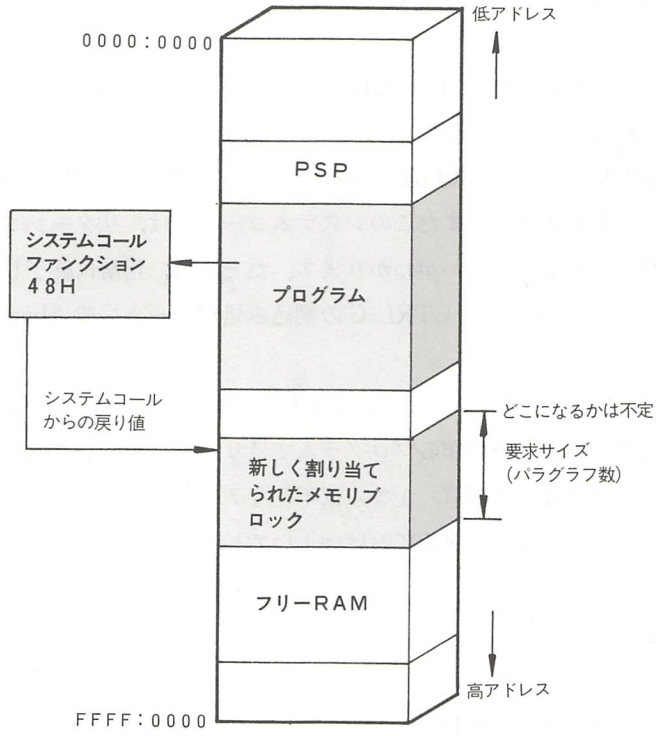


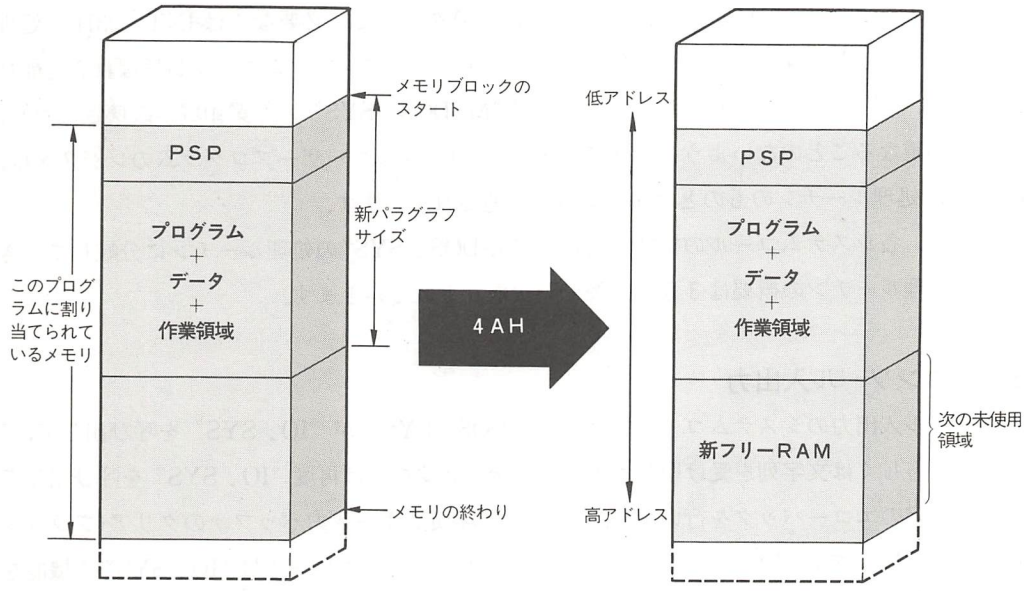
Fig. 3.40　メモリの割当て

ファンクション49Hは，ファンクション48Hのシステムコールによって割り当てたメモリブロックが不要となったとき，それをDOSに返すためのものです．

このシステムコールでは，メモリブロックの先頭アドレスを渡すだけで，返すブロックのサイズは指定する(システム側でわかっている)必要はありませんが，ファンクション48Hのシステムコールで割り当てられていないものを返すことはできません．

ファンクション4AHのシステムコールは，ファンクション48H(または4AH)のシステムコールによって割り当てられたメモリブロックのサイズを変更する(拡大／縮小を行う)ために使われます．ファンクション4BHのシステムコールによって新たに起動されたプロセスは，通常PSPからメモリの最終部までが1つのメモリブロックとなっているため，そのプロセスから新たにプロセスを起動するためにはFig. 3.41に示すようにファンクション4AHのシステムコールを用

いて新プロセスのためにメモリをいくらか解放してやらなければなりません．このシステムコールに必要なパラメータは，もとのメモリブロックの先頭アドレスと残しておきたいメモリのパラグラフサイズです．このサイズがもとのブロックサイズよりも小さい場合にはメモリの解放となり，大きい場合には，さらにメモリを要求していることになります．ですから，ファンクション4BHのシステムコールを使いたいときはPSPの先頭のセグメント・アドレスをメモリブロックの先頭アドレスとしてファンクション4AHのシステムコールに渡し，新しいサイズとしてそのプログラムが動くために必要なパラグラフのサイズを渡します．



Fig．3．41　ファンクション４ＡＨによるメモリサイズの変更

3.8　システムコールとMSDOS．SYSの内部処理

本節では，プログラム中でシステムコールが行われたときに，〝MSDOS．SYS〟が行う次の処理について解説を行います．

(1)システムコールの呼出し
(2)コンソール入出力
(3)ディスクバッファリング
(4)ファイル・ハンドルの設定およびプロセスとの関係
(5)コンフィギュレーション

3.8.1　システムコールの呼出し

プログラム内でシステムコールが使用されると(INT　21H)，プログラムの制御は〝MSDOS．SYS〟のシステムコール処理部へ移行します．ここではまず，AH レジスタにはいっているファンクション番号が正しいかどうかを調べ，正しくなければそのまま IRET で戻り，正しい場合は各レジスタの退避をしておいてから，システムコールの各処理ルーチンへ分岐します．この後は，レジスタをもとに戻し，必要ならば結果をレジスタへセットした後，IRET でもとのプログラムに戻ります．

基本的な処理はこれだけのことなのですが，実際にはもう少し複雑になっています．たとえば，ブレークチェック(CTRL-C が押されているかどうか)を行って，必要ならば INT　23H の処理ルーチンを呼び出します．またこの処理ルーチンの中でさらにシステムコールが呼ばれる可能性があるため，2 レベルのシステムコールによって〝MSDOS．SYS〟の作業領域が破壊されたり，スタックが重なることのないようにできています．このときのユーザープログラムのレジスタは，INT　23H 処理ルーチンのものとは別の領域に退避されています．

この後は，各システムコールの機能に応じた〝MSDOS．SYS〟の処理ルーチンに分岐していきます．各処理ルーチンの概要は 3.8.2 以降で順次取り上げていきます．

3.8.2　コンソール入出力

コンソール入出力のシステムコールでは，〝MSDOS．SYS〟が〝IO．SYS〟を呼び出して，1 文字ずつ，もしくは文字列を受け取ります．この後，必要ならば再度〝IO．SYS〟を呼び出して入力した文字のエコーバックを行います．キー入力の判定，キー入力バッファのクリア(フラッシュ)も〝IO．SYS〟で処理されるため，コンソール入力のシステムコールは〝IO．SYS〟の機能を数個ずつ組み合せることによって実現しています．

例として 1 文字入力の方法を考えてみると，エコーバックの有無，CTRL-C によるブレークの可否，入力の前のバッファフラッシュの有無の組合せで実現しています．この組合せの例ですと，8通りのシステムコールがあることになりますが，実際には8通りもありません．これは単にMS-DOS が CP／M との互換性を考慮したためです．

3.8.3　ファイルとディスクバッファリング

MS-DOS は CP／M 系の DOS であるといわれていますが，ファイルの扱いに関しては両者はまったく異なっています．CP／M で扱うファイルのサイズは，レコード単位(1 レコード＝128 バイト固定)ですが，MS-DOS では，1 バイト単位で扱われ，最大 1G バイトの大きさまで扱うことができます．その結果，ファイルの読出し／書込みの単位(1 回のシステムコールで読出し／書込みのできるバイト数)は，CP／M ではレコード単位(128 の倍数バイト)ですが，MS-DOS では 1 バイト単位でアクセスできるようになっています．

CP／Mにおける1レコードの大きさは，ディスクの物理的フォーマットにより決定されています．これは，CP／M開発当時，ディスクといえば今でいう8インチ片面単密度のもの(1枚あたり256Kバイト)しかなく，このディスクが1セクタ当たり128バイトであったことから，これをもとにして作られたCP／Mが1レコード128バイトとなったのです．

CP／M開発当時ならともかく，最近のディスクは倍密度のものが主流になっており，さらにセクタサイズを大きくする(512，1024バイト／セクタ)ことによってディスク1枚あたりの総容量も1～2割程度アップさせることができます．このため，セクタサイズによってファイルサイズを制約するような方法では，効率的にディスクを使用することができなくなってきています．

MS-DOSではこのことを考慮して，ユーザーがセクタの存在を意識する必要がないように，すべてのファイルの長さやシステムコールではバイト単位でファイルを扱えるようにしています．

実際のディスクはあるまとまったブロック単位でしか読み書きすることしかできません．通常このブロックはセクタ単位であり，128，256，512，1024バイトのうちのいずれかの大きさを持っています．ですから，システムコールでファイルをバイト単位でアクセスするためには，何らかの方法でバイト単位のアクセスをできるようにしなければなりません．たとえば，プログラムでファイルから10バイトのデータを読み込もうとするときに，512バイトも転送されたのではデータの読込み領域の後ろにある502バイトの領域が破壊され，プログラムが暴走してしまいます．このような事態を避けるとともに，データを正しく読み書きする方法として，MS-DOSではディスクのバッファリングを行っています．

○ディスクバッファリング

ディスクバッファリングとは，ファイルの読み書きのバイト数がセクタの倍数でない場合，1セクタを"MSDOS．SYS"内に用意したバッファに転送した後，その中の必要部分のみを指定された領域へ転送することを指します．

つまり，ディスクのバッファリングはファイルがアクセスされるたびに行われるわけではなく，読み書きするファイルのアクセスの位置が，ちょうどセクタの境界からはじまっており，アクセスのサイズがセクタサイズの整数倍のときは，このディスクバッファリングの操作は必要ありません(Fig. 3.42 ①)．アクセスがセクタの境界からはじまり，アクセスのサイズがセクタサイズの整数倍でないときには，最終セクタに対してディスクバッファリングを行う必要があります(同図②)．さらに，ファイル・アクセスがセクタの境界からはじまっていないときは，先頭のセクタに対してもディスクバッファリングを行う必要があります(同図③)．

ディスクバッファリングは，ファイルを読み出すときだけでなく，書き込むときにも行われます．この場合，目的のセクタの内容がいったんバッファに読み出され，その後プログラムの転送領域からバッファへ転送されます．このバッファは，この後すぐにディスクに書き込まれずに，メモリ中にそのまま残っています．そしてこのバッファには，後でディスクに書き込まれなけれ

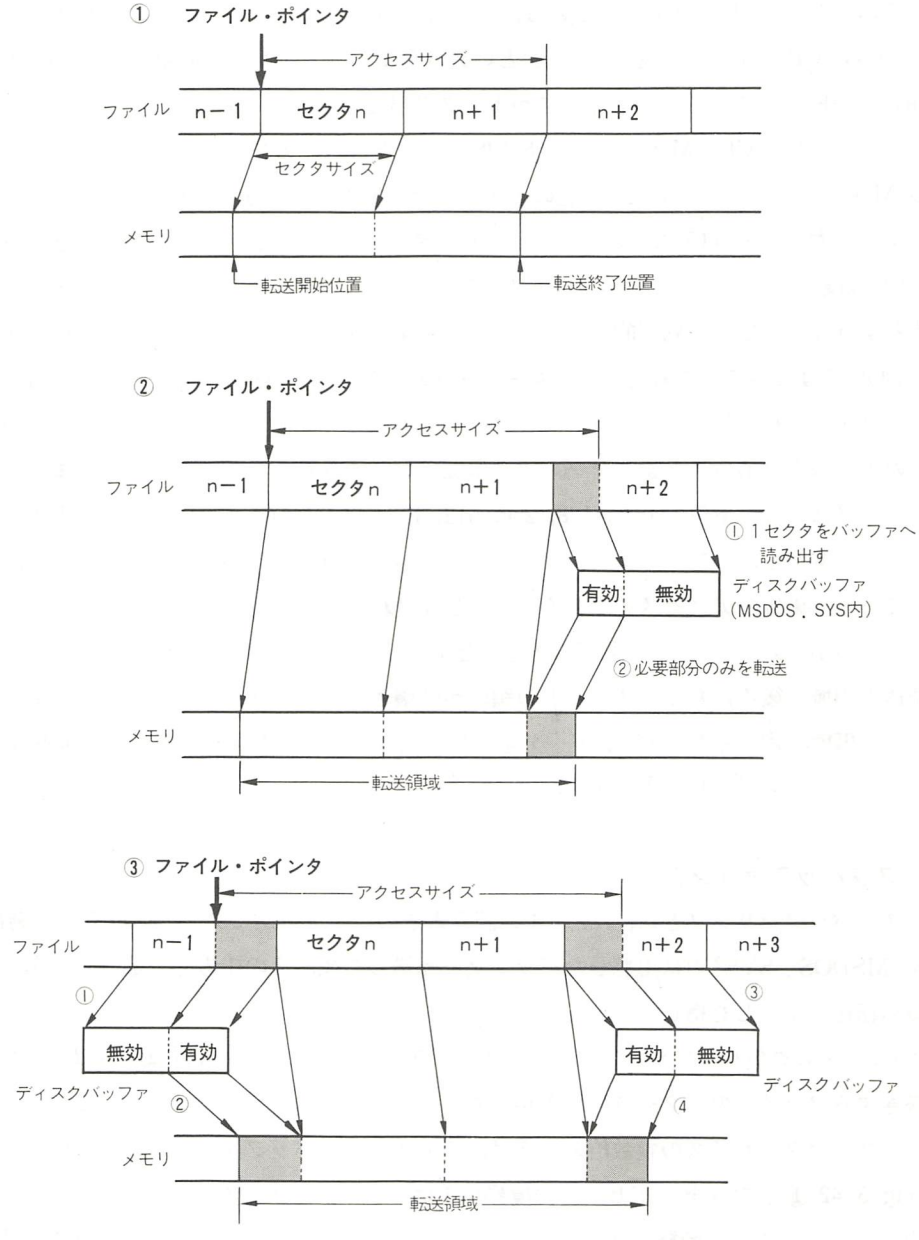
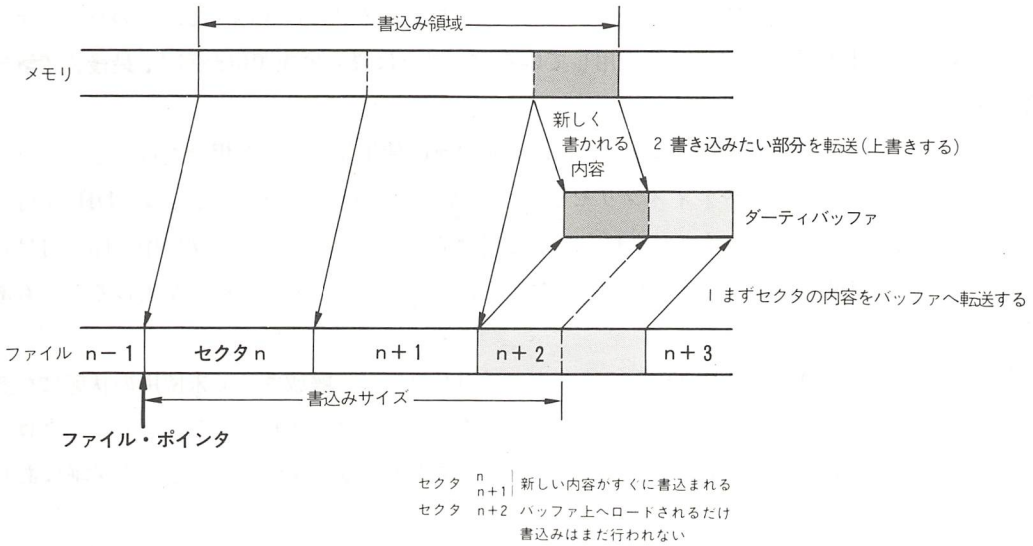


Fig. 3．42　ディスクバッファリング

ばならないという旨の印が付けられます．このようなバッファのことをダーティバッファ(dirty buffer)と呼びます．

ダーティバッファをすぐにディスクに書き込まずにメモリ上に残しておくことは，ディスクバッファリングのもう1つの目的であるディスク・アクセスを最小限に抑えるためです．このよう

にしておけば，次回の書込み時にディスクから読み出してくる必要がなくなります．Fig.3．43の例でいうと(次回の書込みのシステムコールで)，セクタn＋2の内容を読み込んでこなくとも，単にダーティバッファの後半へのデータを転送するだけでセクタn+2への書込み動作は完了します．この場合，最初に読み出したセクタn+2の内容は，すべて上書きされてしまうので，最初の読出しはむだになってしまいますが，これはファイルがシーケンシャル・アクセスで書き込まれているからであり，ファイルにランダム・アクセスを行う場合には，十分意味を持ってきます．ただし，ファイルの書込みに先立ってセクタの読出しを行わなければならないのは，ファイルに上書きする場合(ランダム・アクセスの書込みなど)，あるいはファイルへの追加の場合であり，ファイル作成のシステムコールを行ってファイルに新規に書込む場合には必要ありません．このときのダーティバッファの未使用領域の内容は不定となります．



Fig．3．43　ダーディバッファ

今まで述べた内容は，ファイルに関するディスクバッファリングについてですが，MS-DOSでは，FATやディレクトリのアクセスにもバッファを用いています．ディスク上のFATやディレクトリ領域をメモリに読み込む場合，空いているバッファに読み込まれ，そのバッファには使用中(つまり，バッファの内容が有効であるという意味)の印が付けられるとともに，そのバッファの管理情報として，どのドライブのどのトラック，セクタを保持しているかが記録されます．また，書込みに対しては，もしその必要セクタがバッファ上にあるときには，バッファ内の必要部分を更新して，その後すぐにディスクへ書き込みます．必要とするセクタがバッファにない場合には，空いているバッファへ目的のセクタを読み出し，更新を行ってからディスクへ書き込みま

す．どちらの場合も最終的にはそのバッファが使用中であるという印を付けておくので，ディレクトリやFATをアクセスする場合，バッファにあれば高速に参照することができます．このように，FATやディレクトリなどはファイル・システムにとって重要な部分であるので，つねにディスクの内容を最新のものに更新するようにしています．

○バッファの解放

バッファが割り当てられる場合，空いているバッファがある限りそれらが利用され，なるべくディスクの多くの部分がメモリ上に存在するようにしています．というのは実際にディスクへアクセスする回数が減り，処理速度の向上が望めるからです．しかし，バッファの要求に対してすべてのバッファが使用中である場合には，いずれかのバッファを解放しなければなりません．このとき，解放するバッファの順番は，まず読出しだけに使用されているバッファが解放され，このようなバッファがない場合には，ダーティバッファをディスクに書き込んで，そのバッファを解放します．FATやディレクトリが使用しているバッファは最も優先順位が高く，最後まで解放されずに残っています．

ダーティバッファのディスクへの書込みは，バッファの使用要求が出た場合だけでなく，ファイルがクローズされたときやディスクリセットのシステムコール(ファンクション0DH)を行ったときにも起こります．ファイルをクローズした場合でも，このバッファは使用中の印が付けられているので，再度ファイルを読み直すようなときにはディスクへのアクセスが省けることもあります．

ディスクリセットのシステムコールでは，すべてのバッファは解放されて未使用の状態になるため，バッファのデータはすべて無効となります．また，オープンされているファイルはクローズされずにそのまま残っていますから，ディレクトリやFATの内容が正しいという保証はありません．

○バッファの数

ディスクバッファリングを行えば，頻繁にアクセスされるディスク上のセクタをメモリにおいて，処理速度を向上させることができるのですから，単純に考えればバッファの数さえ増やせば(デフォルトでは2個)効率がよくなるはずです．しかし，バッファのサイズはそのシステムで利用しているディスクのうちで最も大きなセクタサイズを持っているものに合せられます．5インチフロッピーディスクなら512バイト，8インチフロッピーディスクなら1024バイトです．これだけのメモリ領域がバッファの数だけ取られるのですから，あまり多いとメモリ領域が減ってしまいます．また，あまりバッファの数を増やすと，目的のセクタがバッファにあるかどうかの判定に時間がかかるために，ディスクをアクセスするよりも遅くなることがあります．そのため，バッファの数は15～20ぐらいに設定するのが妥当なところでしょう．

○バッファリングとハードウェア

ディスクバッファリングがその威力を発揮するためには，重大な1つの前提が必要です．それは，メモリ上の各バッファの内容とディスク上のセクタの内容が一致していなければならないということです．当然のことのようですが，もしこの状態が維持されないとすれば，つまりディスクが交換されて処理が進められれば交換されたディスクは書込みにより破壊されてしまう可能性があります．

このような事態を避けるために，MS-DOSではディスク・アクセス(ファイルの読出し／書込み時は除く)のたびに，〝MSDOS．SYS〟から〝IO．SYS〟へディスクが交換されたかどうかを調べるコマンドを発行しています．〝IO．SYS〟ではこのコマンドを受けると，最後にディスクがアクセスされてから(物理的にディスクをアクセスしてからこのコマンドが発行された間に)ディスクが交換されたかどうかを調べ，その結果を次の3つに分類して，〝MSDOS．SYS〟へ値を返します．

(1)ディスクは交換されていない
(2)ディスクは交換されている(単にディスクのドアが開けられて，もう一度閉められただけという状態か，本当にディスクが別のものと交換されているという状態のどちらか)
(3)ディスクが交換されたかどうか不明

ハードウェアでディスクのドアが開けられたことを検出できる場合には，(1)あるいは(2)の値が返されますが，そのような機構を持っていないハードウェアのために(3)の値が用意されています．

多くの5インチディスクを装備したシステムでは，ディスクのアクセスがないときにはドライブのモータを止めて消費電力を抑えたり，熱の発生によるディスクの疲労を防いでいるために，このようなドライブでは，ディスクが交換されたことを検出する方法を持っていません．そのため，〝IO．SYS〟はつねに(3)の値を返しています．

どうしてモータを止めてしまうとドアが開けられたかどうか検出できないのかについて，少し説明しておきましょう．

普通，ディスクがドライブにはいっていてモータが回転している場合，1秒につき5回(5インチディスク)または6回(8インチディスク)のインデックスパルスが発生しており，ドライブ側ではこのパルスをもとにREADY信号をONにします．インデックスパルスはフロッピーディスクに開けられたインデックスホールを横切る小穴を光センサーでつねに監視することにより検出されるため，FDC側ではこのREADY信号がONならばドライブにディスクがはいっていると判断してアクセスを行い，OFFならばドライブの準備ができていないものとみなします．ただしOFF状態のうちには，本当にドライブの中にディスクがはいっていないという状態と，ディスクがはいっているがモータが回っていないためインデックスパルスが発生しないので見かけ上NOT READYとなっているという2つの状態があります．

ですからつねにモータが回っているようなシステムでは，ドライブからのREADY信号のON／OFFを監視していれば，ディスクがはいっているか((1)の状態)，それとも交換されたかどうか((2)の状態)がわかります．FDCによってはこの2つの状態を自動的に検出してくれるものもあります．

しかし，非アクセス時にドライブのモータを止めてしまうようなシステムでは，ディスクがはいっていないためにREADY信号がOFFなのか，それとも単にモータが止まっていてREADY信号がOFFなのかがわからないため，(3)を返すしか方法がありません．

それでは次に，上記の3つの値を受け取ったときに，〝MSDOS．SYS〟がどのような動作を行うかを見ていきます．

(1)ディスクが交換されていない場合

バッファとディスクの内容は一致しているはずなので，〝MSDOS．SYS〟が管理しているバッファはすべて有効であるとみなされます．つまり，余計なディスク・アクセスは行われず，最も効果的にプログラムが実行されます．

(2)ディスクが交換された場合

バッファ(ダーティバッファも含む)の内容はすべて無効となり，未使用状態となります．

(3)ディスクが交換されたかどうか不明の場合

バッファの状態により2通りの場合に分かれます．バッファの中にダーティバッファがある場合には，ディスクは交換されていないとみなされ，(1)と同様に扱われます．ダーティバッファがない場合，(2)と同様にバッファの内容はすべて無効となり，未使用状態になります．

3つの値のうちどの値が返されるかによって，MS-DOSの応答速度は劇的に変化します．8インチディスクのシステムでは，多くの場合(1)か(2)を判定できるため，ディスクバッファリングの効果が最も有効に働きます．

ハードディスクやRAMディスクの場合には，つねに(1)の値が返されるために，デバイス自体の高速性と相まって，MS-DOSを最も快適に使用することができます．残念ながら，ほとんどの5インチディスクをベースにしたMS-DOSではハードウェアの制約によって〝IO．SYS〟が(3)の値しか返さないために，ディスクバッファリングの効果が活かされません．

ここに述べたのは一般的なMS-DOSシステムの話ですから，5インチシステムながら(1)，(2)の値を返すものもありますし，逆に8インチシステムでもディスクの交換が検出できないために，非常に応答速度の遅いシステムもあります．また，1つのシステムでも，2種類以上のディスクをサポートしている場合は，その種類によって返す値が変わるシステムもあります．バッファはドライブごとに管理されているので，(2)や(3)の値が返された場合には，その値を返したドライブごとにバッファをフラッシュしますから，1つのシステムでも，8インチディスクは速いが5インチ

ディスクは遅いということもあります．

○システムの判別

現在利用しているMS-DOSシステムが，速いシステムであるかどうかを見分けるには，次のようにして簡単に行うことができます．

まず，〝CONFIG．SYS〟ファイルに次に示す行を入れて，バッファの数を10個に設定します（別に10個でなくてもよいのですが，このくらいが適当です）．

```
BUFFERS = 10
```

次にリセットボタンを押してシステムをリブートします．これでバッファが10個確保されました．ここで，ディスクのドアを開けるかCTRL-Cを押してバッファをすべてフラッシュさせます．この後，1回DIRコマンドを実行してディスクのディレクトリを見ます．このときディスクがアクセスされたことを確認してください．たいていのシステムでは，ディスクのヘッドの動作音がしてLEDが光るはずです．これでディスクのディレクトリとFATの領域がバッファに読み込まれました．

さて，次が重要なのですが，もう一度DIRコマンドでさきほどと同じディスクの同じディレクトリを見てください．このときディスクをまったくアクセスしないようであれば，そのシステムはバッファを有効に活用しています．つまり，ディスクの交換を判別しているので，バッファを多くとればMS-DOSの性能をいかんなく発揮できます．

2度目のDIRコマンドの実行でもディスクをアクセスするようであれば，残念ながらそのシステムはバッファを有効に使用していません．いくらバッファを多くしてもメモリのむだになりますから，デフォルトの2ないし5個ぐらいに設定しておけばよいでしょう．しかし，このシステムに接続できる他の種類のドライブはこうではないかもしれません．ハードディスクやRAMディスクが使用できるのであれば，バッファを増やす価値があります．そうでない場合には，ディレクトリを深くしたり，多くのファイルやバッチコマンドを作るというようなことは避けた方がよいでしょう．特にバッチコマンドでは，バッチファイルの中の1行を処理するたびに，ディスクへ次の1行を読みに行きますから，不必要なREM文を入れたり，ラベルを付けたり，GOTOコマンドで飛ばすようにしておくと非常に速度が落ちます．試しに，VERとかECHO, VERIFY, BREAK, CLS, PATHなどのディスクをまったくアクセスしないコマンドばかりがはいったバッチファイルを，深いディレクトリに作って実行させてみてください．ディスクバッファリングの効果が非常によく現れます．

3.8.4　システム用FCBとファイル・ハンドル

FCBを使用する場合，扱えるファイルの数に制限がありませんが，ファイル・ハンドルを使用

する場合には，1プロセスあたり最大20個までしか扱うことができません．この20という数字は，標準でオープンされているファイル・ハンドル(5個)も含めた数ですから，プログラム中に新たにオープンできるファイルの数は(標準ファイル・ハンドルをクローズしなければ)，15個ということになります．ただし，"CONFIG．SYS"ファイル中で，オープンできるファイルの数を指定しなかった場合，15個ではなく5個しかオープンできないので，標準入出力などと合せると，最大10個のファイル(ファイル・ハンドル)が扱えます．

ファイル・ハンドルで扱うファイルの数を増やすと(CONFIG．SYSファイル中で指定する)，1つのファイルにつき40バイトほどユーザー用領域が減ります(システム領域が増える)．この領域は，ファイル・ハンドル用のFCBに使われており，"MSDOS．SYS"がファイル・ハンドル用に確保しているシステムテーブルに設定されます．

MS-DOSはシステムのブート時に，"CONFIG．SYS"の中から使用するファイルの数を読み取り，その数だけ"MSDOS．SYS"内に内部的に使用するための(ファイル・ハンドル用)FCBを用意します．これらのFCBには0からはじまる一連の番号を付け，以降この番号でこれらのシステム用FCBを参照します．

次に標準ファイル・ハンドル用に，CON，PRN，AUXの3つのデバイスをオープンしておきます．この後，"MSDOS．SYS"は"COMMAND．COM"に制御を移すときに20個のフィールドを持つテーブルを作り，先頭から順に(0〜4)さきほどのFCBの番号を設定して標準ファイル・ハンドルのテーブルとします(Fig. 3.44)．

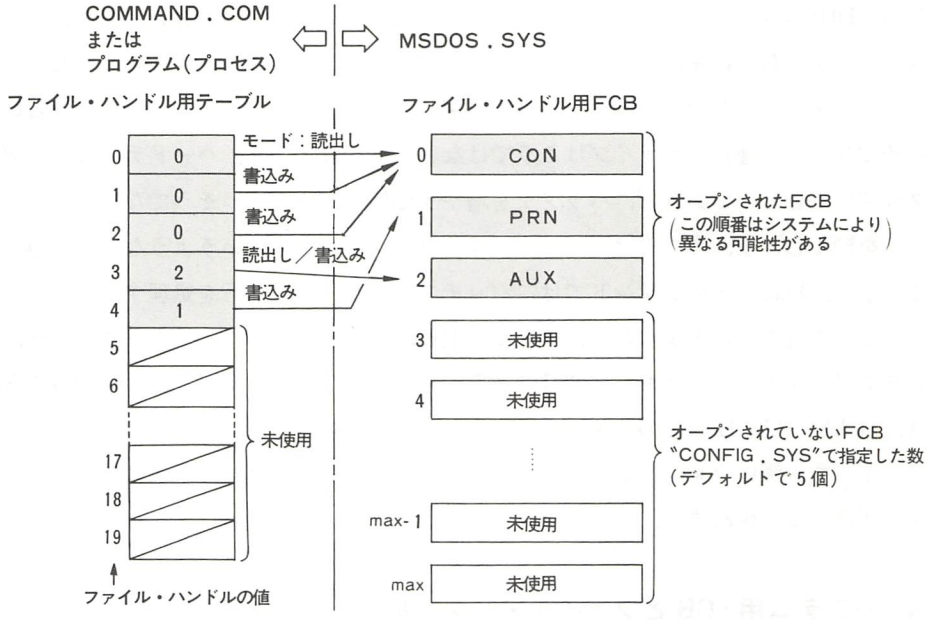


Fig. 3．44　システム用FCBとファイル・ハンドル

Fig. 3.44 からもわかるように，"COMMAND．COM" 用に用意されたテーブルのインデックス(添字)が，ファイル・ハンドルの値となっています．たとえば，ファイル・ハンドルが2ということはテーブルの先頭から3番目のものを指しており，それがCONに割り当てられているため，ファイル・ハンドル2への書込みは，コンソールへの出力となります．

この後，"COMMAND．COM"がファイルをいくつかオープンしたとすると，それらのファイルに対して返されるファイル・ハンドルは5からはじまることになります(Fig. 3.45)．

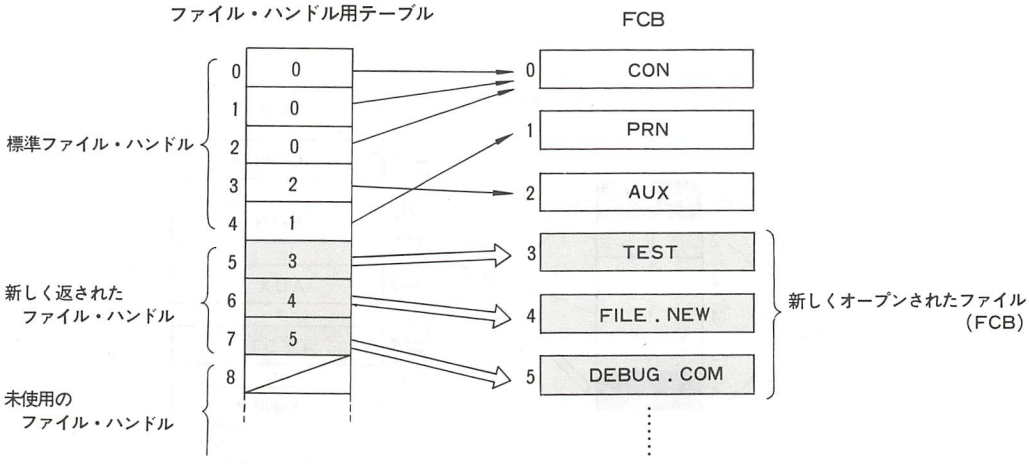


Fig. 3 .45　オープンされたファイル

次にファイルをクローズした場合を考えてみます．今の Fig. 3.45 の状態からファイル・ハンドル1と6をクローズしてみると，Fig. 3.46 のようになります．

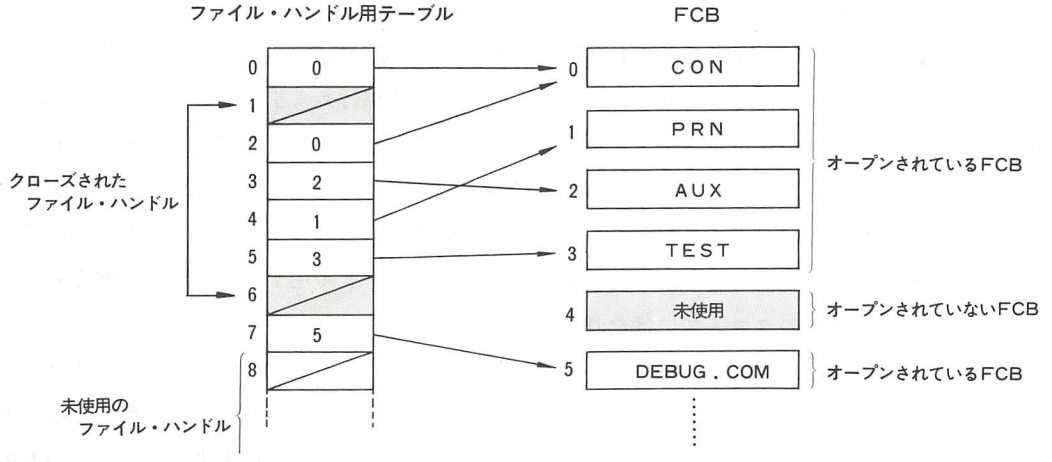


Fig. 3 .46　ファイルのクローズ

この後ファイルをオープンすると，ファイル•ハンドルの値としては1が返され，次に6，8の順になります．またFCBはまず4が使われ，後は6から先が使われます．

ファンクション45Hのシステムコールでは，ファイル•ハンドルのコピーを行うことができますが，これは指定したファイル・ハンドルのテーブルの内容を空いているファイル・ハンドルのテーブルへコピーするだけです．例として今のFig.3.46の状態から，ファイル•ハンドル5を1回コピーして，さらにファイル•ハンドル2を1回コピーすると，結果はFig.3.47のようになります．このとき最初のコピーではファイル•ハンドル1が返され，2回目のコピーでは6が結果として返されます．

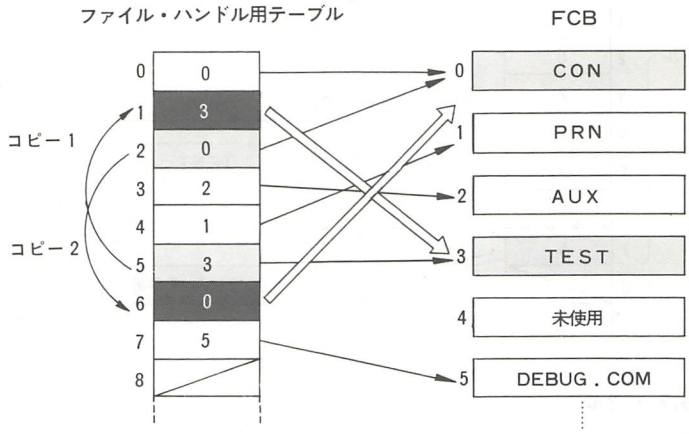
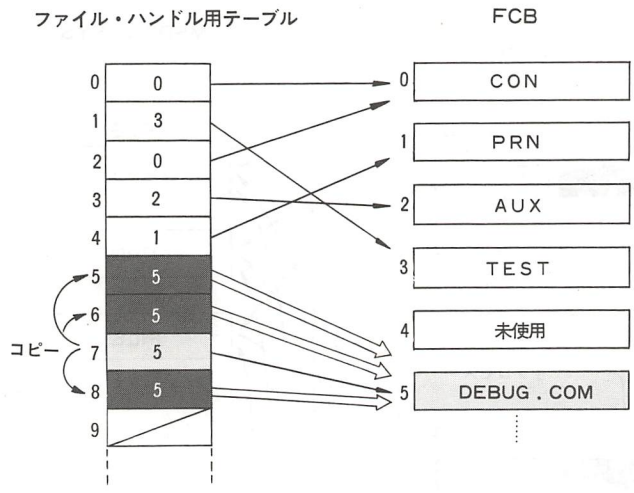


Fig.3.47　ファイル・ハンドルのコピー

このFig.3.47からもわかるように，このシステムコールでは単にファイル•ハンドルをコピーするだけで，新たに同じ内容を持ったFCBを用意するわけではないので，どちらのファイル•ハンドルを用いてもまったく同じようにファイルへのアクセスが行えます．このとき，どちらか一方のファイル・ハンドルを用いてファイルをクローズしても，もう1つのファイル・ハンドルの方は正しくFCBを指しているために，そのまま継続してファイルのアクセスを行うことができます．つまり，2つ以上のファイル•ハンドルが同じFCBを指しているとき，どれか1つのファイル・ハンドルをクローズしても，それは単にファイル・ハンドル用テーブルから削除されるだけで，本当のファイルのクローズにはなりません．

ファンクション46Hもファイル•ハンドルのコピーを行いますが，コピー先のファイル•ハンドルの番号を指定する点がファンクション45Hのシステムコールと異なります．コピー先のファイル・ハンドルが使用中の場合，そのファイル・ハンドルはクローズされます．さきほどのFig.3.47の状態から，さらにファイル•ハンドル7を5，6，8へコピーした例がFig.3.48の状態です．



Fig．3．48　ファイル・ハンドルのコピーの強制

なお，ファイル・ハンドル用のテーブルのフィールドが20しかないので，ファイル・ハンドルの値が20以上となることはありません．これ以上ファイルをオープンしようとすると，エラーとなります．

3.8.5　ファイル・ハンドルとプロセスの関係

ここでは1つのプロセスにおいてファイルがオープンされたとき，そのファイルが子プロセスあるいは孫プロセスにおいてどのように扱われるのか，いい換えれば，プロセス間で制御が移る場合，〝MSDOS．SYS〟はファイル・ハンドルをどのように扱うかについて解説していきます．

ファンクション4BHのシステムコールにより子プロセスを起動する場合，子プロセスは親プロセスの環境やオープンしているファイルを受け継ぎます．ファイルを受け継ぐというのは，親プロセスが使用していた有効なファイル・ハンドルが，そのまますべて継続して利用できるということです．ただし，逆は成り立ちません．たとえば，親プロセスでオープンしたファイルのファイル・ハンドルを，子プロセスでクローズして別のファイルをオープンしたとしても，親プロセスに戻ったときには子プロセスがクローズする前の状態に戻ります．

前項でファイル・ハンドルを記憶しておくテーブルについて述べましたが，実はこのテーブルはすべてのプロセスにつき1つずつ用意されるもので，新しいプロセスが作られるとき，親プロセスのテーブルの内容がすべて子プロセス用のテーブルにコピーされます．つまり〝MSDOS．SYS〟のシステム用FCBを，2つ以上のプロセスが共有しているために，子プロセスで親から受け継いだファイルをクローズしても，親プロセスのファイル・ハンドルが有効なために実際にはファイルはクローズされず，単に子プロセスのテーブルから削除されるだけです(Fig．3．49)．

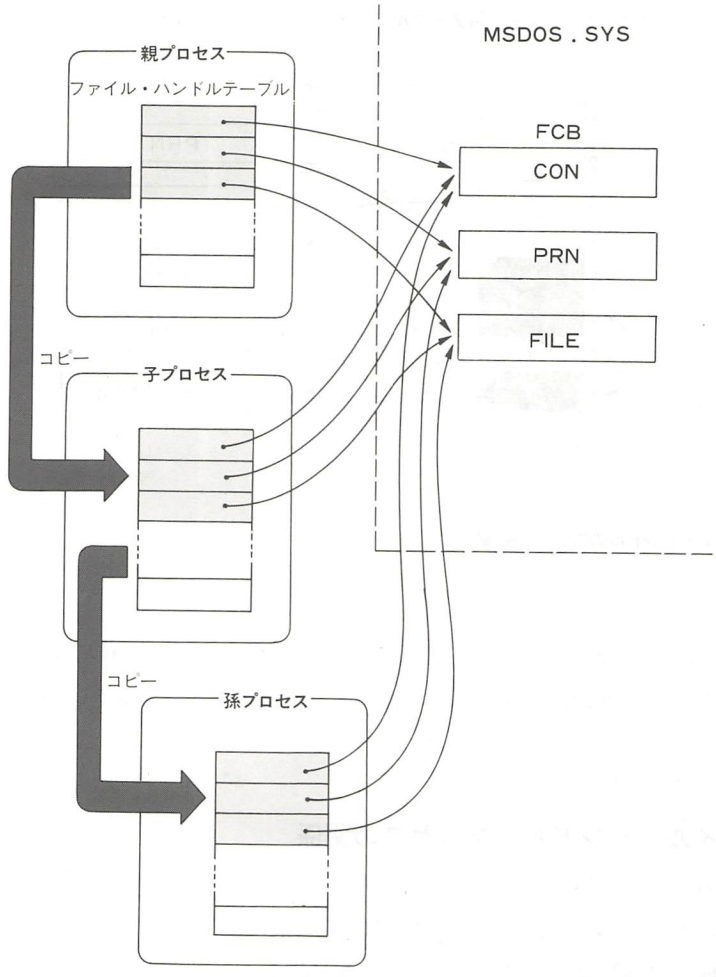


Fig. 3．49　ファイル・ハンドルの継承

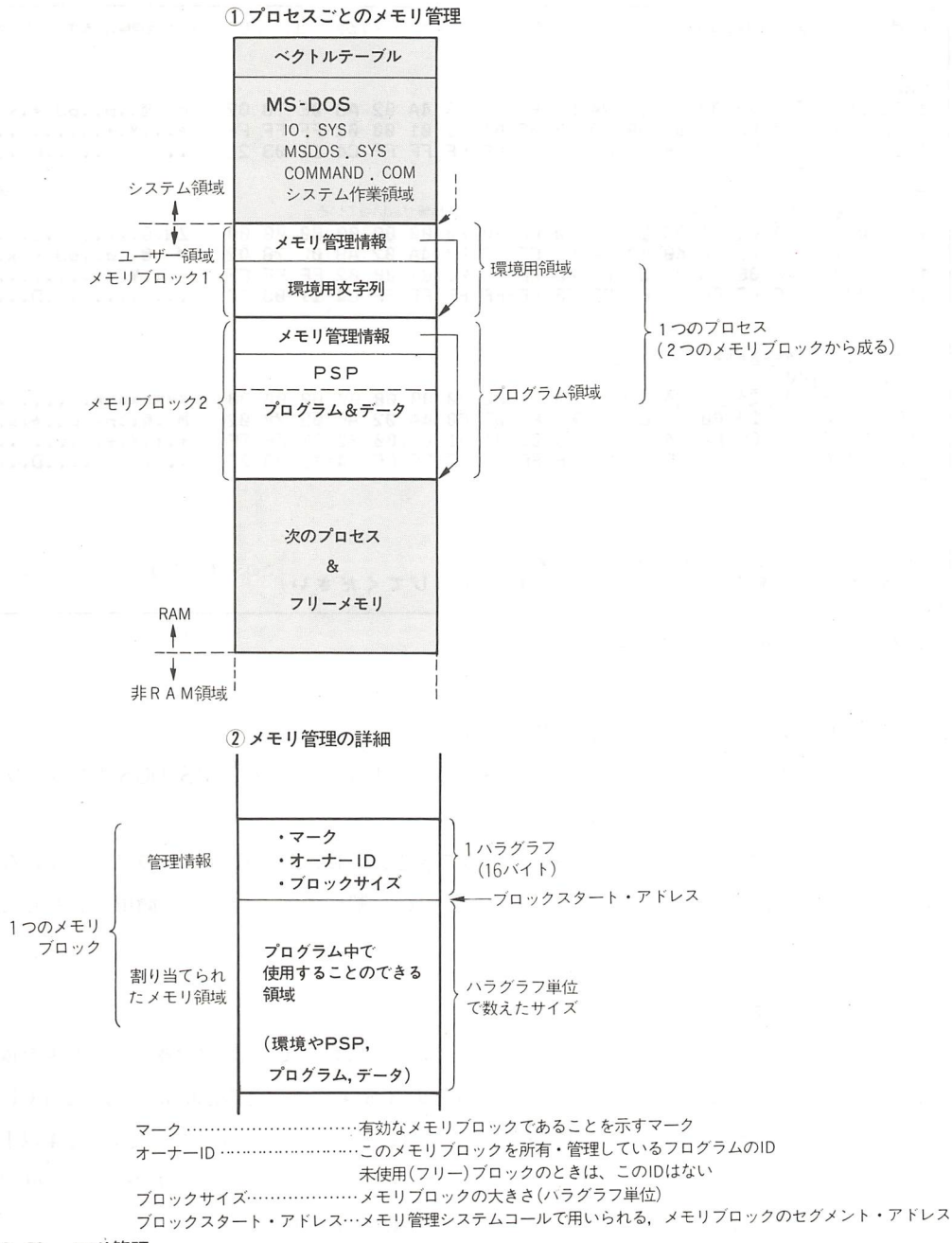
ファイル・ハンドルがこのように受け継がれているとすれば，親プロセスは子プロセスの標準入出力(ハンドル 0，1)を制御するだけでなく，標準エラー出力(2)，標準補助入出力(3)，標準プリンタ出力(4)のすべてを制御できるはずですが，"COMMAND．COM"ではこれらの標準ファイル・ハンドルについてはサポートしていません．

3.8.6　メモリ管理

メモリ管理の必然性はいまさらいうまでもなく，複数のプロセスがお互いの領域を破壊しないために行われるものです．これは，将来 MS-DOS がマルチタスクの DOS となったときのためだけでなく，現在の ver. 2.0 においても非常に重要な役割を果たしています．

MS-DOS ではメモリをパラグラフ単位で管理しており，各メモリブロックの先頭にそのブロ

ック(パラグラフの集合)に関する情報を置き，それらのブロックをポインタでリンクすることによってメモリ管理を行っています(Fig.3.50)．ブロックに関する情報とは，そのブロックが使用中か否か，使用中だとすればそのブロックを所有しているプロセスはどれか，ブロックのサイズ(パラグラフの数)などを指します．



Fig．3．50　メモリ管理

Fig. 3．51 では，DEBUG を使用して故意にメモリの管理情報を破壊した例を示しています．このように，割り当てられたブロックの直前のパラグラフを書き換えてしまうと正常なメモリ管理ができなくなり，〝COMMAND．COM〟はメモリアロケーションエラーを表示して止まってしまいます．

```
A>debug command.com ↵   ……DEBUGを起動する．ファイルをロードするのは，正しいPSP(とメモリ管理情報)を作らせるため
-
-
-d0 130 ↵   ……PSPの先頭30Hバイト
11C8:0000  CD 20 00 40 00 9A F0 FF-0D F0 4A 02 AB 0E 78 02   M .@..p..pJ.+.x.
11C8:0010  AB 0E BA 04 D9 0D AB 0E-01 01 01 00 02 FF FF FF   +.:.Y.+.........
11C8:0020  FF FF FF FF FF FF FF FF-FF FF FF FF C4 11 03 2C   ............D..,
-          ↓PSP
-
-d11c7:0 140 ↵   ……PSPの1つ手前のパラグラフにメモリ管理情報がはいっている
11C7:0000  5A C8 11 38 2E 00 00 00-00 00 00 00 00 00 00 00   ZH.8............
11C7:0010  CD 20 00 40 00 9A F0 FF-0D F0 4A 02 AB 0E 78 02   M .@..p..pJ.+.x.
11C7:0020  AB 0E BA 04 D9 0D AB 0E-01 01 01 00 02 FF FF FF   +.:.Y.+.........
11C7:0030  FF FF FF FF FF FF FF FF-FF FF FF FF C4 11 03 2C   ............D..,
-          ↓PSP
-
-e11c7:0 '********' ↵   ……管理情報をでたらめなデータで壊す
-d11c7:0 140 ↵
11C7:0000  2A 2A 2A 2A 2A 2A 2A 2A-00 00 00 00 00 00 00 00   ********........
11C7:0010  CD 20 00 40 00 9A F0 FF-0D F0 4A 02 AB 0E 78 02   M .@..p..pJ.+.x.
11C7:0020  AB 0E BA 04 D9 0D AB 0E-01 01 01 00 02 FF FF FF   +.:.Y.+.........
11C7:0030  FF FF FF FF FF FF FF FF-FF FF FF FF C4 11 03 2C   ............D..,
-          └壊れている
-
-q ↵   ……〝COMMAND．COM〟へ戻る

メモリの アロケーションエラーです
COMMAND.COM をロードできません．リセットしてください   } このメッセージを出力して止まってしまう
```

Fig. 3．51　DEBUGによるメモリの破壊

3.8.7　コンフィギュレーションファイル

〝CONFIG．SYS〟ファイルは，MS-DOS のブート時に参照され，MS-DOS 内のシステムによる細かな設定を行うためにあります．

このファイルの内容はエディタで簡単に編集できるように，ASCII コードのファイルになっています．以下では，使用することのできるコマンドの一般形を示しながら，簡単に説明しておきます．

FILES ＝ファイル数

これは，ファイル・ハンドルを用いたシステムコールにおいて，最大何個のファイルを扱うかを決めるものです．別のいい方をすれば，さきほどの 3.8.4 で述べた内部的に使用する FCB の数を決めるためのコマンドです．この指定がないときは，デフォルトで 5 となっており，4 以下を指定しても無視されます．最大数は 99 ファイルですが，実際には 1 プロセスあたり最大 20 ファイルしか扱えないため，通常は 15〜20 ぐらいが妥当な設定数です．

BUFFERS ＝バッファ数

3.8.3で述べたバッファの数を設定するもので，デフォルトでは2が設定されています．最大99バッファまで許されますが，あまり多いとメモリの空き領域が狭まりますし，バッファ検索に時間がかかるために逆にディスク・アクセスよりも遅くなったりします．また，いくらバッファを増やしても，ほとんど意味のないシステムもあるので，3.8.3の内容を参考にしてシステムに最適のバッファ数を決めてください．

BREAK ＝{ON または OFF}

〝COMMAND．COM〞のBREAKコマンドとまったく同じ働きをします．デフォルト時はOFFとなっています．

DEVICE ＝デバイス・ドライバのファイル名［オプション列］

新しいデバイスをシステムに付加するためのものです．詳しくは4章を参照してください．

SHELL ＝コマンド・プロセッサのパス名［オプション列］

これはシステムが使用するコマンド・プロセッサを指定するコマンドです．デフォルトは〝COMMAND．COM〞であり，ブートディスクのルート・ディレクトリからロードされます．しかし，このコマンドを使うと他のディレクトリやドライブからでも〝COMMAND．COM〞もしくは自分で作った新しいコマンド・プロセッサをロードすることができます．

〝COMMAND．COM〞をコマンドプロセッサとして使うときは，／Pスイッチを付けて，メモリ中に常駐するように指定しておかなければなりません．また，オプションとしてディレクトリを表すパス名を付加しておくと，〝COMMAND．COM〞を再ロードする必要が生じた場合（COMMAND．COMの非常駐部が破壊されたとき），指定したディレクトリから再ロードさせることができます．デフォルトは，カレント・ドライブのルート・ディレクトリとなっています．この再ロードを行うパス名は，環境変数〝COMSPEC＝〞で指定され，〝COMMAND．COM〞はそのパスから非常駐部を読み込みます．このため〝COMSPEC＝〞を破壊すると〝COMMAND．COM〞の再ロードができなくなるので注意して下さい．この環境変数〝COMSPEC＝〞は〝COMMAND．COM〞の起動時のオプションで設定することができます．

たとえば，次のようにSHELLを指定するとコマンド・プロセッサ〝COMMAND．COM〞をディレクトリ〝C：¥BIN〞からロードして実行せよという指示であり，再ロードも同じディレクトリから行えという指定にもなっています．

```
SHELL＝C：¥BIN¥COMMAND．COM　C：¥BIN ／P
```

COUNTRY ＝国番号

国番号としては，1(米国)か81(日本)のみが許されます．これにより，MS-DOSが保存している国別情報が指定したものと変わります．デフォルトでは，日本メーカーのMS-DOSでは当然81です．なお，これによって国を指定しても，表向きはDIRコマンドで表示される日付のフォーマットが変わるぐらいです．

4章 IO.SYS

〝IO.SYS〞の機能の概要についてはすでに1章で述べてあるので，ここでは〝IO.SYS〞の中心的な機能であるデバイスの制御の詳細について触れていきます．とくに，〝IO.SYS〞の大半はデバイスを制御するためのデバイス・ドライバの集まりなので，解説はこのデバイス・ドライバの構造や機能を中心に行います．

デバイス・ドライバ自体は多分に機種に依存しており，また，その作成もMS-DOSを供給しているメーカーで行われるので，その内部動作について詳しく紹介しても無意味ですが，デバイス・ドライバの呼出し方の手順や指定された各コマンドで行うべき動作についてはどの機種でも共通ですので，解説はその共通部分について行うことにします．

MS-DOSではユーザーがデバイス・ドライバを作成して，簡単にシステムに付加できるという特徴があるので，この解説は単に〝IO.SYS〞の説明だけではなく，デバイス・ドライバを作成するためのガイドにもなっています．

4.1 IO.SYSとデバイス・ドライバ

〝IO.SYS〞の働きは，〝MSDOS.SYS〞の制御のもとで，MS-DOSシステムに接続されている各種のデバイス(周辺装置)を管理することですが，この管理するためのプログラムのことを，デバイス・ドライバと呼びます．

〝IO.SYS〞には，デバイス・ドライバとして，コンソールとの入出力，プリンタへの出力，補助装置(RS-232Cなど)，リアルタイムクロック(カレンダ＋時計)，ディスクとの入出力を行うための5つのデバイス・ドライバが含まれています．ただし，これらのデバイス・ドライバがすべて最初から備わっているとは限りません．〝IO.SYS〞内のデバイス・ドライバの作成は，マイクロソフト社ではなくMS-DOSの供給を受けているメーカー側で行われるため，若干の変更が加わります．たとえば，RS-232Cやリアルタイムクロックがオプションとなっている機種では，そのためのデバイス・ドライバがオプションとなっていることがあります．また，同じ種類のデバイスでもメーカーによっては仕様が異なってくることがあります．その差が最も顕著に現れるのは，コンソールへの出力に関する仕様です．A社では多くのエスケープシーケンスが用意されているが，B社のものでは数個しか用意されておらず，文字に対する属性(反転，ブリンク，下線など)が少ない，また表示速度に関しても違いがあるなど，各社によってまちまちです．

さらにディスク・ドライブにおいても，同じ5インチ両面倍密度のディスク・ドライブを使用

しているにもかかわらず，あるメーカーの仕様のものは，片面倍密度のディスク(160Kバイト)が読み書きできないということもあります．

以上は，〝IO．SYS〞に含まれているデフォルトのデバイス・ドライバの話ですが，MS-DOSでは，ユーザーが新たにデバイス・ドライバをシステムに追加することもできます．さらに，既存のデバイス・ドライバの仕様が不満であれば，それを新しいデバイス・ドライバと置き換えることも可能です．

デバイス・ドライバの追加，修正を行うためには，システム側(IO．SYS)に対する修正は必要ありません．単に新しいデバイス・ドライバを作成し，それを〝CONFIG．SYS〞ファイルに登録するだけで済んでしまいます．これがMS-DOS ver. 2.0の大きな特長の1つにもなっています．

さて，これ以後の解説はデバイス・ドライバを中心に行いますが，とくに断わらない限り，その説明は〝IO．SYS〞に組込みのデバイス・ドライバとユーザーが作成するデバイス・ドライバの両方に当てはまります．両者にあまり大きな差がないのは，MS-DOSの拡張性に対する基本設計がよかったからだといえます．

4.1.1　ブロック型とキャラクタ型デバイス

MS-DOSで扱うデバイスは，その入出力に対する扱いの違いから，ブロック型とキャラクタ型のデバイスの2種類に分けられています．前者は，ディスクのように一度にあるまとまった量のデータを読み書きできるランダム・アクセスが可能なデバイスのことを指します．これに対しキャラクタ型デバイスは，一度に1バイトずつ読み書きするデバイスのことを指し，通常ランダム・アクセスはできません．後者には，コンソール(スクリーン＋キーボード)やプリンタ，RS-232Cなどが該当します．

キャラクタ型のうちの特殊なものとして，クロック型のデバイスがありますが，通常このデバイスに対するデバイス・ドライバはメーカーの方から供給済みであり，ユーザーが新たに作成することはないでしょう．

デバイス・ドライバは，各デバイスにつき1つずつ用意されています(しなければならない)．ただし，ブロック型デバイスに対しては，1つのデバイス・ドライバで複数のドライブをサポートすることができます(たとえば，1つで〝A：〞，〝B：〞，〝C：〞，〝D：〞の4ドライブを扱う)．これに対し，キャラクタ型のデバイス・ドライバでは，CON，PRN，AUX，(CLOCK)のそれぞれに対して1つずつ(論理的に区別できればよい)デバイス・ドライバを用意します．

4.1.2　デバイス・ドライバの構造

デバイス・ドライバは，デバイスを制御するためのプログラム部分とデバイスヘッダの2つの部分から成っています．基本的にはプログラム部分さえあればデバイスは制御できるわけですが，

MS-DOS側でデバイス・ドライバを管理するためには，いくつかの情報が必要です．たとえば，そのデバイス・ドライバはブロック型のデバイスを制御するのか，キャラクタ型のデバイスを制御するのかにより，扱われ方が大きく違ってきます．また，ブロック型のデバイスの場合は，そのデバイス・ドライバで何台のデバイスを扱うのかという情報や，キャラクタ型のデバイスの場合は，そのデバイスに対してどんな名前でアクセスするのかといった情報も必要となります．そのほかにも，デバイス・ドライバを呼び出すために必要なプログラムのスタート・アドレスもわかっていなければなりません．

そこでデバイス・ドライバには，プログラムだけではなくこれら一連の情報も持たせることにしてあります．MS-DOSは，この情報を参照してからそのデバイス・ドライバに対する扱いを決定するために，これらの情報はすばやく参照できなければなりません．そのことを考慮した結果，各デバイス・ドライバの先頭にデバイスヘッダと呼ばれる固定長の領域が取られ，その中にさきほど述べたデバイス・ドライバに関する種々の情報が入れられています．そしてプログラム部分は，そのヘッダの後に続くというようになっています(Fig. 4.1)．なお，デバイスヘッダの詳細については後述します．

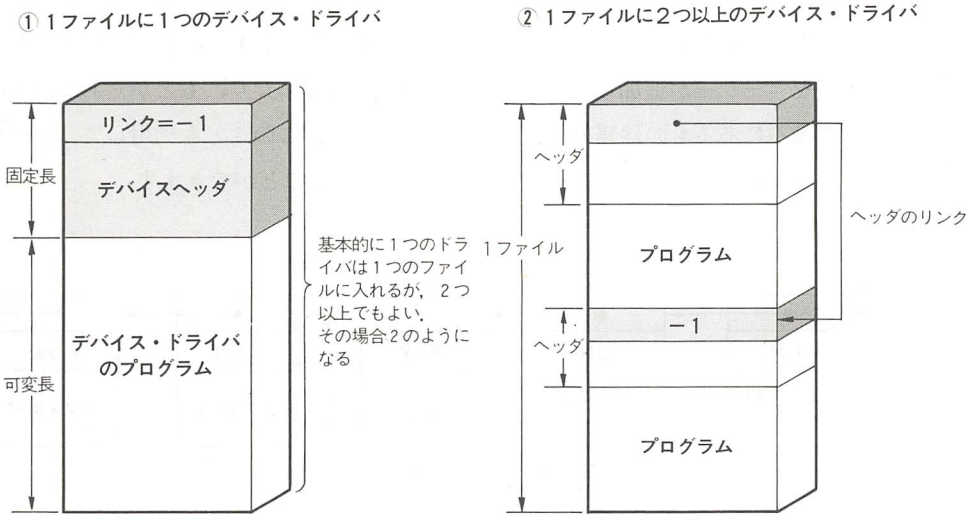
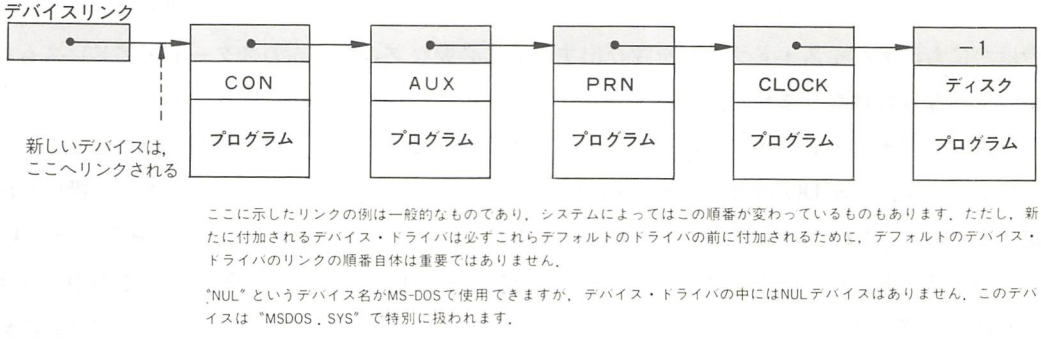


Fig. 4.1　デバイス・ドライバの構成

4.1.3　デバイス・ドライバのリンク

デバイス・ドライバには，さきほど述べたようにメーカーが作成して〝IO.SYS〟に最初から組み込んであるものと，ユーザーが後から追加するものの2種類があり，これらのデバイス・ドライバは，1本のチェインでリンクされてシステムで管理されます．このうち，デフォルトのデバ

イス・ドライバは，Fig. 4.2のようにリンクされます．なお，このリンクのためのポインタはデバイスヘッダの中に用意されています．



ここに示したリンクの例は一般的なものであり，システムによってはこの順番が変わっているものもあります．ただし，新たに付加されるデバイス・ドライバは必ずこれらデフォルトのドライバの前に付加されるために，デフォルトのデバイス・ドライバのリンクの順番自体は重要ではありません．

"NUL"というデバイス名がMS-DOSで使用できますが，デバイス・ドライバの中にはNULデバイスはありません．このデバイスは"MSDOS．SYS"で特別に扱われます．

Fig. 4．2　デフォルトのデバイス・ドライバのリンク

ユーザーが作成したデバイス・ドライバは，このデフォルトのデバイス・ドライバ群の先頭にリンクされます．

キャラクタ型デバイスは名前によって識別されているため，Fig. 4.3に示すようにまったく同じ名前のデバイスをユーザーが追加した場合，追加した方が有効になり，もとからあったものやリンクチェインの後ろにあるものは無効となります．ですから，このことを利用して，ユーザーが新しいCONやAUXなどのデバイス・ドライバと置き換えることができます．

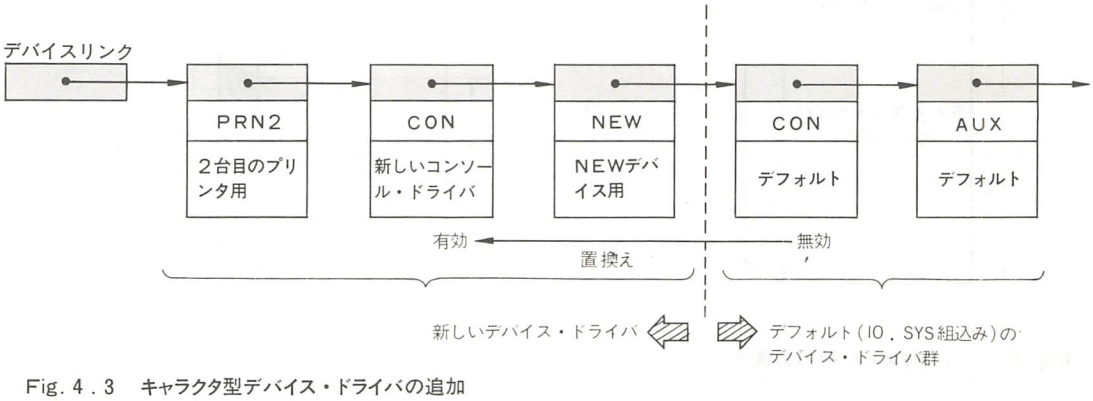


Fig. 4．3　キャラクタ型デバイス・ドライバの追加

ブロック型のデバイス・ドライバも，ユーザーが新たに作成したものをシステムに登録することができます．ただし，キャラクタ型のデバイス・ドライバと違って，新しく登録されたものはシステムへの追加とみなされます．つまり，デフォルトのデバイス・ドライバと置き換えることはできないのです．ブロック型のデバイスはドライブ名("A：", "B：", "C："……)で参照され

るため，複数のドライバがあった場合，デバイス・ドライバのリンクされる順番と1つのドライバでいくつのドライブがサポートされるかにより，各ドライブのドライブ名が決まります．MS-DOSでは複数のブロック型デバイス・ドライバがある場合，リンクの先頭の方から順に〝A：〟，〝B：〟，〝C：〟……と割り当てていきます．たとえば，あるドライバ(仮にALPHAとしておきます．あくまでもこれは仮の名称であり，実際に名前があるわけではありません)が3台のドライブをサポートし，別のドライバ(BETAとする)が2台のドライブをサポートするとします．このとき，2つのドライバのリンクされる順により，各ドライブの名前はFig. 4.4のように変わります．

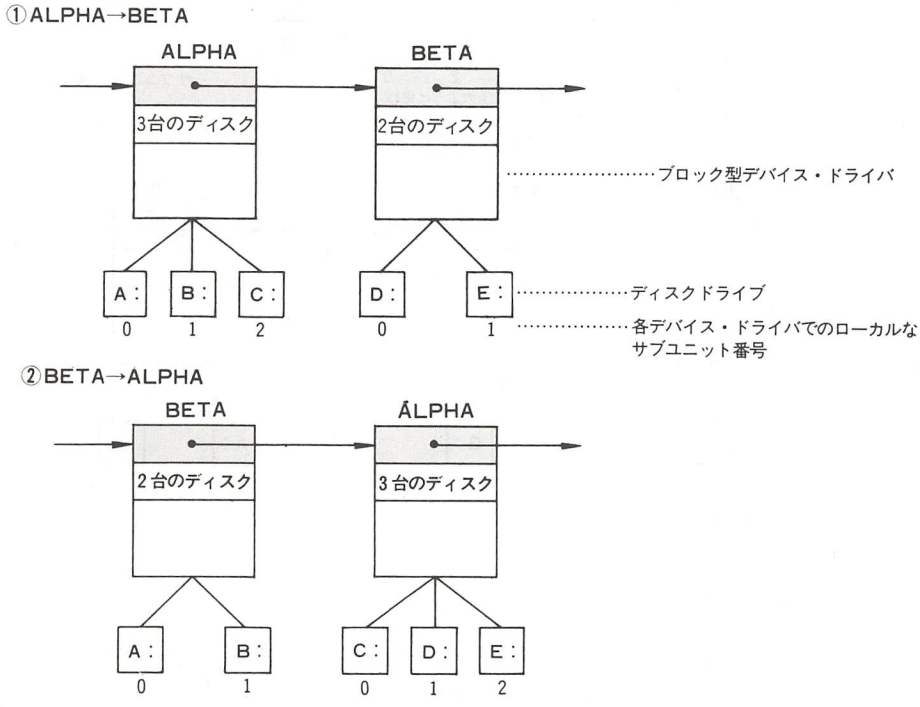
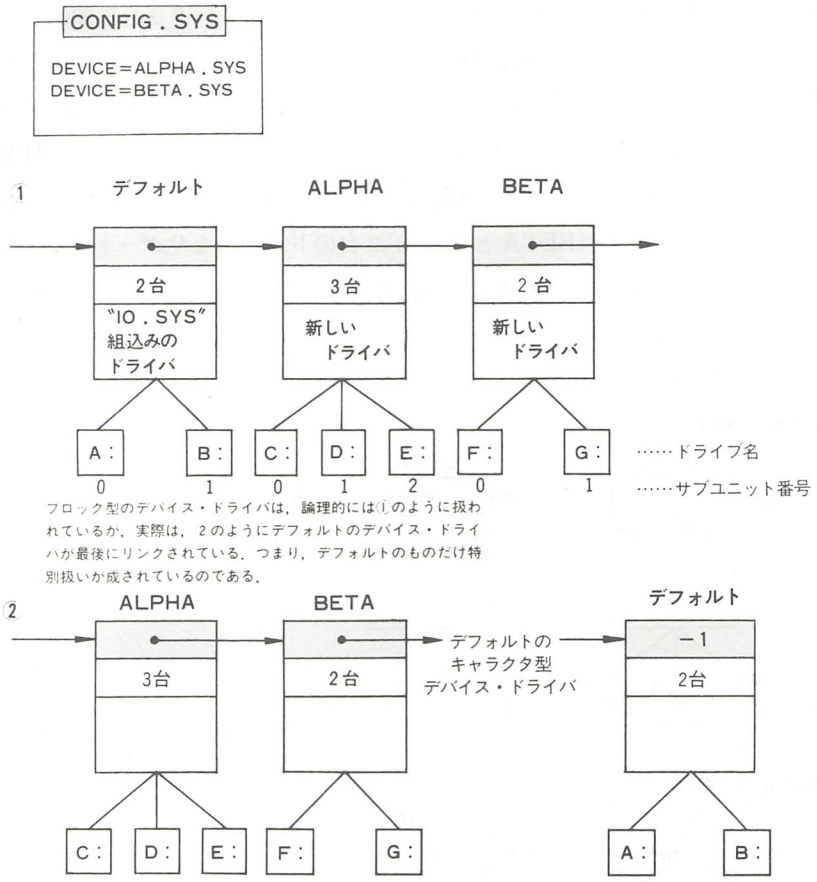


Fig. 4.4　ブロック型デバイスのドライブ名

ユーザーが新規に作成したブロック型のデバイス・ドライバは，登録した順に(CONFIG.SYS中で現れた順に)既存の(IO.SYSに組み込まれている)デバイス・ドライバの後へ追加されていくため(論理的に後として扱われるが，実際のリンクでは前へ追加される)，ドライブ名は既存のドライブ名の次のものから使用されます．たとえば，あるシステムで〝A：〟と〝B：〟の2台のドライブが最初から使用されているとします．このとき，さきほどのALPHAとBETAの各デバイス・ドライバをこの順に〝CONFIG.SYS〟に登録したとすると，各ドライブの名前はFig. 4.5のようになります．



Fig．4．5　ブロック型デバイス・ドライバの追加

4.1.4　デバイスヘッダ

4.1.2で述べたように，MS-DOSがデバイス・ドライバを管理するための情報は，デバイスヘッダにあらかじめ設定されており，その情報を参照することによってデバイス・ドライバの管理が行われています．デバイスヘッダの構成はTable 4.1のようになっており，実際のソース・プログラムではFig. 4.6のように記述されます．

Table　4．1　デバイス・ヘッダの構成

デバイスヘッダ

2ワード	次のデバイスへのポインタ，普通は FFFFH（－1）にセットしておく	固定長
1ワード	デバイスの属性	
1ワード	デバイスストラテジエントリ・ポイント	
1ワード	デバイス割込みエントリ・ポイント	
8バイト	デバイス名またはユニット数	

```
;
; device header for
;   character device  (device name is 'CON')
;
console dw      -1,-1           ;pointer area for device link
        dw      8000h+10011b    ;con_in & con_out & special
        dw      strategy        ;strategy entry point
        dw      interrupt       ;interrupt entry point
        db      'CON    '       ;device name
;
strategy:

;  code for strategy entry point

interrupt:

;  code for interrupt entry point
```

Fig. 4.6　デバイス・ヘッダの実例

○リンク情報

システムで使用可能なデバイス・ドライバは，さきほど述べたように"IO.SYS"で1本のリンクにして保持されており，必要なデバイスはリンクを頭からたどることにより検索されます．このためのリンク情報が，デバイス・ドライバの先頭2ワード(セグメント＋オフセット)に用意されており，デフォルトのデバイス・ドライバではこのフィールドは，次のデバイスのヘッダ・アドレスを指していますが，ユーザーが新たに追加するデバイスの場合，このフィールドには2つのFFFFH(−1)を入れておきます．こうしておけば"IO.SYS"によって自動的に次のデバイス(ヘッダ)を指すようにデバイス・ドライバのロード時にセットされます．

○属性

次の1ワード(属性ワード)はFig. 4.7に示すように，このデバイスのタイプを示しています．

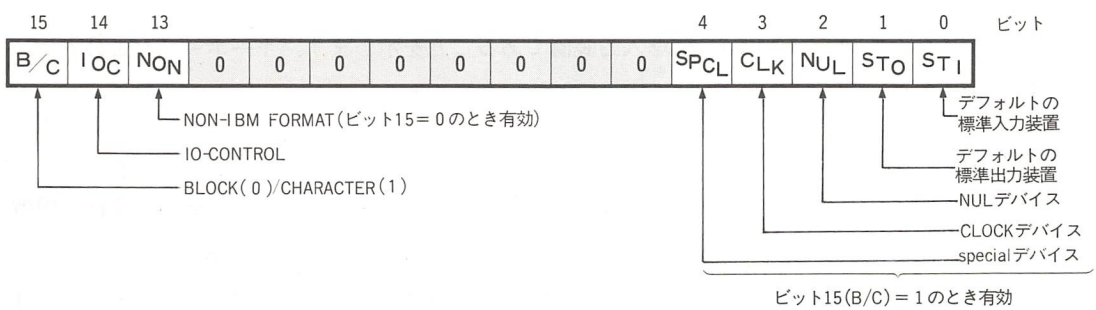


Fig. 4.7　属性ワードの説明

ビット15(BLOCK／CHARACTER)

このデバイスの種類を表しており，このビットが1にセットされているときはキャラクタ型デバイスであり，0のときはブロック型デバイスとなっています．

ビット14(IOCTL)

ファンクション44Hのシステムコールにおけるデバイスに対するI／Oコントロールが可能かどうかを示します．1のとき可能であり，0のとき不可能となっています．I／Oコントロールとは，前章で述べたように，デバイスとの入出力ではなく，デバイス・ドライバとの入出力を行うことを指します．これにより，RS-232Cの仕様やプリンタの種類の設定など，デバイスとそのドライバ自身の状態設定，把握が行えます．

ビット13(NON-IBM FORMAT)

ブロック型のデバイスにおいてのみ意味があります．このビットが1にセットされていると，このブロック型のデバイスで扱うディスクはIBM-PCフォーマットのMS-DOSディスク(MS-DOS標準フォーマットを満たしているディスク)ではないものとみなされます．通常MS-DOSで扱うディスクには，どの機種でも他機種のMS-DOSのディスクを読み書きできるようにFATやディレクトリ，データ領域の位置や大きさ，その他ディスク全体やセクタの大きさなどの情報がすべてディスクの種類ごとに決まっていますが，RAMディスクや大容量のハードディスクなど，標準で仕様が決まっていないものについては，デバイス・ドライバから〝MSDOS．SYS〟へそのディスクに関する種々のパラメータの値を知らせてやらなければなりません．そのために用意されているのがこのビット13です．なお，MS-DOS標準ディスクフォーマットについては，5章を参照してください．

属性ワードのビット0からビット4は，このデバイスがキャラクタ型デバイスである場合にのみ意味を持ちます．

ビット0(STI)

このデバイスが(デフォルトの)標準入力装置であることを示します．キーボードがこのデバイスに相当します．

ビット1(STO)

このデバイスが(デフォルトの)標準出力装置であることを示します．スクリーンがこれに相当します．

〝IO．SYS〟に含まれているデバイス・ドライバのうち，CONデバイスはビット0とビット1がすでに1にセットされているため，システムブート時には，CONデバイスがデフォルトの標準入出力装置となっています．

ビット2(NUL)

システムが現在NULデバイスとして使っているかどうかを調べるために存在します．デバイス・ドライバ作成時には0にしておかなければなりません．

ビット3(CLOCK)

そのデバイスがクロック・デバイスの場合1にセットしておきます．

ビット4(SPECIAL)

特殊デバイスに対して用いられるビット(special bit)です．このビットは，キャラクタ型デバイスのうち，とくにCONデバイスと深いかかわりがあります．このビットが1のデバイスは，CONデバイスであり，出力に対して，INT　29Hルーチンを使うことができます．MS-DOS ver. 2.0のデバイス・ドライバにおける入出力は，リクエストヘッダと2つのエントリ・ポイントによるデバイス・ドライバへのI／Oリクエスト(詳細は後述)によって行われるため，ver. 1.25のときのJMPテーブルによる方法に比べて，たいへん処理速度が劣っています．つまり，画面出力が遅くなっているのです．ですから，大量の文字データを一度に画面へ出力するときには，コンソールをRAWモードにして，CONファイルへまとめ書きするようにすると出力速度が上がります．しかし，このようにして効果があるのは，あくまで大量の文字(列)を出力するときだけであり，1バイトずつの出力ではファンクション01H～0CHのシステムコールを用いた場合より遅くなってしまいます．そこで新たに用意されたのがINT　29Hによる文字出力です．ALレジスタに文字を入れて，INT　29Hを実行することにより画面に文字が出力されます．しかし，CTRL-SやCTRL-Cはいっさい無視されます．このINT　29Hによる文字出力はver. 1.25と同じ程度の出力速度があるのですが，special bitが1のCONデバイスにしか適用できませんし，MS-DOSの将来のバーションでは使えなくなる可能性があります．また，このspecial bitは，システムの中でただ1つのデバイス・ドライバだけが1にセットすることができます．なお，CONのspecial bitが1かどうかを調べるには，ファンクション44Hのシステムコールが使えます．このビットが0のデバイスやシステムでINT　29Hのコールを行っても無意味です．

○エントリ・ポイント

デバイスヘッダのうち，属性の次には後述する2つのエントリ・ポイントを示すアドレスがはいっています．1つはストラテジエントリ・ポイント(strategy)であり，もう1つは割込みエントリ・ポイント(interrupt)です．これらのエントリ・ポイントは，"MSDOS．SYS"から順次呼び出され，I／Oリクエストが成されます．

○デバイス名

デバイスヘッダの最後のフィールドは，デバイス名のために使用されます．ただし，このデバイスのタイプが何であるかによって，このフィールドの長さ(有効長)が変わります．

キャラクタ型デバイスの場合，このフィールドにはそのデバイス名がはいります．デバイス名は長さ8文字から成るASCII文字列で表されますが，8文字に満たない分については空白が詰められます(デバイス名は左詰め)．また，このデバイス名を表すために使用できる文字は，一般のファイル名として許されているものです．

デバイスがブロック型デバイスの場合，このデバイス名フィールドは，最初の1バイトのみが有効であり，2バイト目以降は使用されません(プログラム領域として使ってよい)．ブロック型のデバイス用のデバイス・ドライバでは，1つのドライバで複数のデバイス(ドライブ)をサポートするのが普通ですが，このフィールドには，その制御しているデバイス数を入れておきます．たとえば，あるデバイス・ドライバが2台の5インチドライブと2台の8インチドライブを扱うのであれば，このフィールドの値は04Hとなります．

4.2　デバイス・ドライバの呼出し

MS-DOSでは，デバイス・ドライバに対してI／Oリクエストを行う際に，コマンドパケット(command packet)と2つのエントリ・ポイントを用いるという方法を取っています．これは，従来の同程度のDOSが用いていた，レジスタによってパラメータを渡しJMP命令によってデバイス・ドライバを呼び出すといった方法と比べると，かなり進歩しているといえるでしょう．

コマンドパケットとは，デバイス・ドライバに対するI／Oリクエストの種類(コマンド)と必要なパラメータを格納したメモリ中の領域のことであり，大きさ(パケットサイズ)はコマンドにより異なりますが，13～22バイトとなっています．またコマンドパケットは，デバイス・ドライバ側へパラメータを渡すだけでなく，要求された動作が正しく行われたか否かの，I／O動作の結果をデバイス・ドライバを呼び出したプログラム(MSDOS．SYS)へ返すためにも使われます．つまり，MS-DOSではレジスタによるパラメータや結果を渡すという方法はいっさい取っておらず，呼出し側とデバイス・ドライバ間の情報交換はすべてコマンドパケットを介して行われています．

また，デバイス・ドライバを呼び出すのに2つのエントリ・ポイントを用意しているのもMS-DOSのデバイス・ドライバの特徴です．現在の処理の内容から考えれば，エントリ・ポイントは1つで十分なのですが，そうなっていないのには深いわけがあります．その事情の説明は後にして，まず先に，コマンドパケットと2つのエントリ・ポイントによるI／Oリクエストの動作の概略について説明しておきます．

4.2.1　I／Oリクエスト処理過程

まず最初に，I／Oリクエストを行う側(MSDOS．SYS)は，コマンドパケットを作成します．この中にはコマンドの種類を表すコード(13種類のうち1つ)や，それに伴うパラメータ類などを入れておきます．たとえば，ディスクから読込みを行いたいとすると，リードコマンド(4)やどの

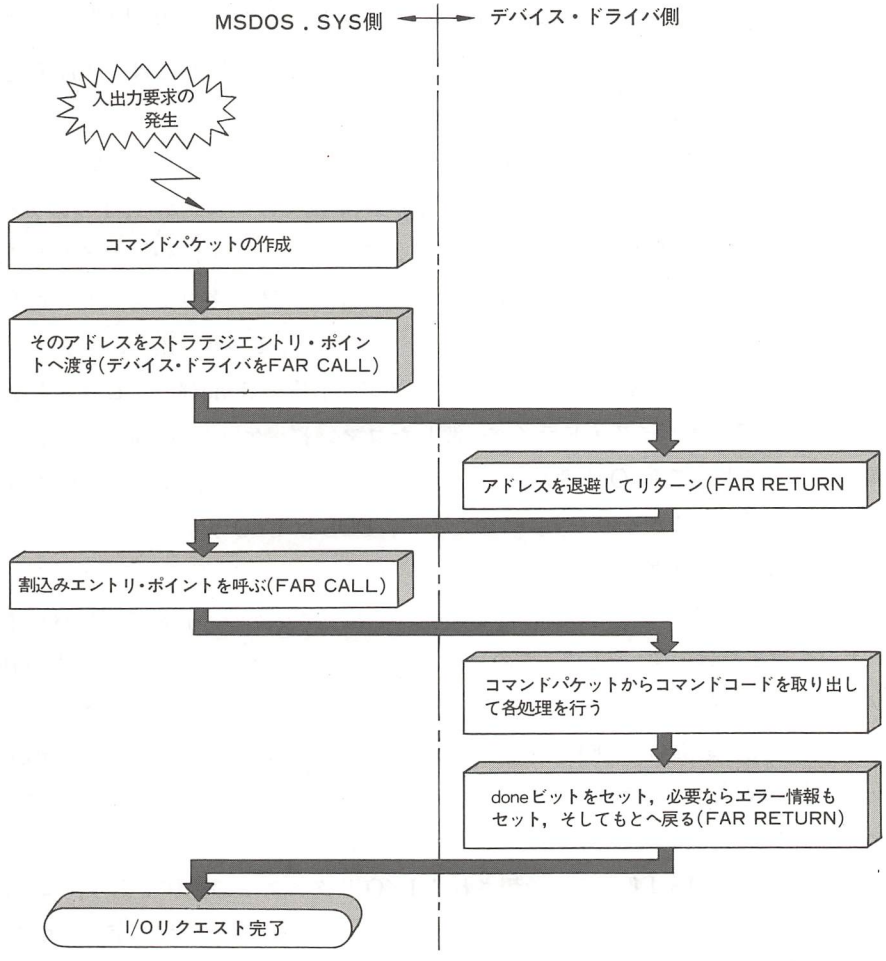
ドライブのどのセクタから何セクタをメモリのどこへ転送するかをコマンドパケットの所定の位置にセットします．

準備ができたら〝MSDOS．SYS〟は，コマンドパケットの先頭アドレスをES：BXレジスタに入れて，デバイス・ドライバのストラテジエントリ・ポイントをFAR CALLします．

デバイス・ドライバ側では，ESとBXレジスタをデバイス・ドライバ内のどこか固定領域へ格納して，後でいつでもコマンドパケットを参照できるようにした後，FAR RETURNを行って，いったんデバイス・ドライバからもとのプログラム(MSDOS．SYS)へ戻ります．

この後，〝MSDOS．SYS〟はすぐに，デバイス・ドライバの割込みエントリ・ポイントをFAR CALLします．

デバイス・ドライバ側では，さきほど格納しておいたコマンドパケットのスタート・アドレスをもとに，コマンドパケットを取り出します．次に，コマンドのコードを取り出して処理の種類



Fig．4．8　I／Oリクエストの処理過程

を決め，それに応じて各処理ルーチンを呼び出します．この後，各処理の結果に応じてコマンドパケット内のステータスワードにエラーがあったか否か，エラーがあったとしたらそれはどのようなエラーか，などの情報をセットします．また，値を返す必要のあるコマンドなら，値をコマンドパケット内の所定の位置へセットします．以上ですべての処理は終わったので，FAR RETURN でもとのプログラムへ戻ります．以上の過程を示したものが Fig. 4.8 です．

4.2.2 〝コマンドパケットと2つのエントリポイント〟の持つ意味

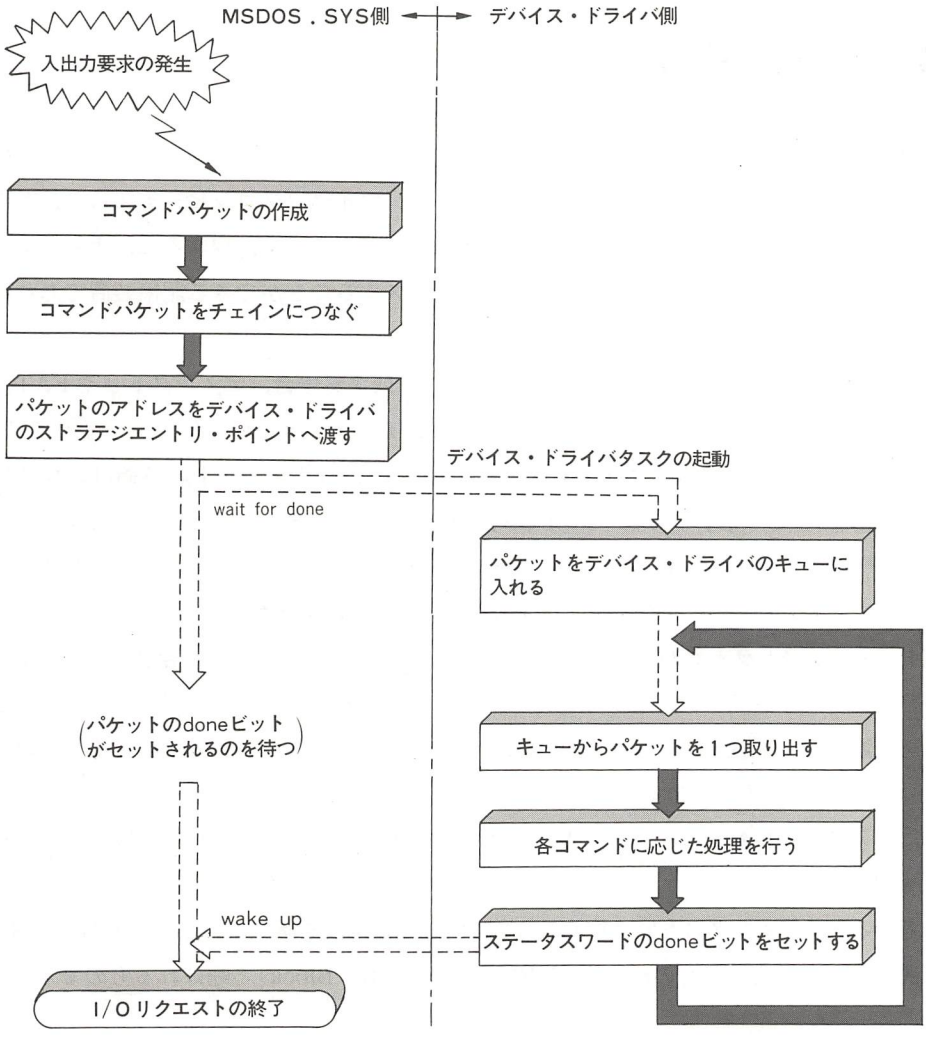
コマンドパケットを用いた入出力動作の概略を述べてきましたが，いっけんしたところ，レジスタで各パラメータをやりとりする方法と比べて，何ら利点が見られないばかりか，むしろその処理の繁雑さが目につきます．しかし，コマンドパケットを用いることに関しては，処理の繁雑さやスピードに関する議論はあまり意味がありません．それよりも，コマンドパケットと2つのエントリ・ポイントによるI／Oリクエストの方法の持つ意味について考える方が重要です．

MS-DOS のI／Oリクエストの方法は，実は将来のマルチタスク版の MS-DOS のことを考慮しており，マルチタスクになったときでもI／Oリクエストの方法に関しては大きな変更を加えなくても済むようにしてあるのです．

マルチタスクの DOS のもとでは，I／Oリクエストは非同期に発生します．つまり，デバイス・ドライバがあるリクエストを処理中かどうかには関係なく，別々のタスクから，1つのデバイス・ドライバに対してI／Oリクエストが発生するのです．このとき，デバイス・ドライバ側では，同時に2つ以上のリクエストを処理することはできないので，いったん各リクエストを到着順にキュー(待行列)に入れておき，キューから順にコマンドパケットを1つ取り出し，リクエストの処理を行います．そして，処理が終わったら，ステータスワード中の done(終了)ビットをセットしておきます．DOS 側ではデバイス・ドライバへ渡したコマンドパケットの done ビットを一定時間ごとに調べることによってI／O処理が完了したことを知り，それにより待たされていたタスクの実行を再開することができるのです．このために，DOS はI／Oリクエストのコマンドパケットをチェインでつないで管理します．

DOS がデバイス・ドライバを呼ぶときは，単にコマンドパケットのアドレスをデバイス・ストラテジルーチンへ渡すだけです．そして，このコマンドパケットのステータスワード中の done ビットがセットされるのを待っています．一方，ストラテジルーチンは，与えられたコマンドパケットをキューの最後尾に追加して DOS へ制御を移してしまいます．このキューの先頭からコマンドパケットを1つ取り出して処理を行うのは，デバイスの割込みエントリ・ポイントで示されるルーチンの仕事です．

以上がマルチタスク版 MS-DOS での予想されるI／Oリクエストの動作であり，図にすると Fig. 4.9 のようになります．しかし，これはまだ存在しないものなので，かならずこうなるという保証はありません．



タスクの起動やwait，wake upなどは，OSのタスクスケジューラの仕事であり，I/Oリクエストを発行するところやデバイス・ドライバ側では，あまり関知しないでよい

Fig．4．9　マルチタスク時のI／Oリクエスト

今述べたことはマルチタスク版になったときの話であり，MS-DOS ver. 2.0のI／Oリクエストではこんなことは行われていません．I／Oリクエストは，一度に１つのデバイス・ドライバでしか実行されませんし，DOSは処理が終わるまで次へ進みません．ですから，ストラテジルーチンでは，単に渡されたコマンドパケットのアドレスをどこか固定領域へ格納した後，すみやかにDOSへ戻らなければなりません．DOSは戻ってきた後，すぐデバイス・ドライバのもう１つのエントリ・ポイント，つまり割込みエントリ・ポイントで示されるルーチンを呼び出します．ですからこのルーチンでは，さきほど格納したコマンドパケットのアドレスをもとに，コマンドを取り出し，そのI／Oリクエストを処理します．そして，すべての処理が終わってからdoneビットをセットしてDOSへ戻らなければなりません．

4.3　BPBとメディアチェック

MS-DOSで扱うことのできるディスクは，5インチや8インチのフロッピーディスクだけでなく，ハードディスクやコンパクトフロッピーディスク，マイクロフロッピーディスク，さらには磁気バブルやRAMディスクのようなものまで扱うことができます．要するに，ディスクのように，あるまとまったブロック単位でランダムにアクセスすることができる記憶装置であればどのようなものでもかまわないわけです．

このようなディスクに対してデバイス・ドライバを作成する場合，ディレクトリやFAT，データ領域を作成し，それを管理するのは"MSDOS．SYS"が行うため，デバイス・ドライバではFATやディレクトリの情報に関しては意識することはありません(ディスク上の各領域の詳細については，5章参照)．しかし，いかに"MSDOS．SYS"でも，今どのドライブにどんなタイプのディスク(メディア)がセットされているのかがわかっていなければ，ファイルをアクセスすることができません．つまり，同じ5インチのディスクでも，片面か両面か，8セクタ／トラックか9セクタ／トラック，倍トラックかそうでないかにより，8通りものメディアが存在しているので，いったいどのタイプのメディアがドライブにセットされているかを正しく把握しておかないと，ディスクを破壊する可能性があります．これは，8インチのディスクに対しても同じことがいえます(片面か両面かなどの違いがある)．

また，ディスクの種類が5インチと8インチの2つだけに限定されるわけではなく，ハードディスクやRAMディスク，また，これから新しく登場するであろうと思われる未知のデバイスのことも考えると，ディスクの形態に捉われない，何らかの手段を講じておかなければなりません．その解決策として考えられたのが，BPB(BIOS Parameter Block)と呼ばれるものです．

4.3.1　BPB

BPBとは，ディスクに関する諸元を設定したメモリ領域のことであり，Table 4.2のような内容を持ち，ソース・プログラムではFig. 4.10のように記述されます(CP／MのDPBに相当する)．

Table　4．2　BPB

1ワード	1セクタのバイト数
1バイト	1アロケーション単位当たりのセクタ数
1ワード	予備のセクタ数
1バイト	FATの数
1ワード	ディレクトリ数
1ワード	論理イメージのセクタ数
1バイト	メディア・ディスクリプタバイト
1ワード	FATセクタ数

```
;
; BPB for 5 inch, single side, double density, 9 sectors
;
        dw      512     ;bytes per sector
        db      1       ;sectors per allocation unit
        dw      1       ;reserved sectors
        db      2       ;number of FATs
        dw      64      ;number of root directory entries
        dw      9*40    ;number of sectors in logical image
        db      0fch    ;media descriptor byte
        dw      2       ;number of sectors per FAT
```

Fig. 4.10　BPBの例

この図で示したように，8インチ，5インチといった区別は含まれていません．単にFATやディレクトリの大きさなどが記録されているだけであり，"MSDOS.SYS"ではこのテーブルを参照してファイルへのアクセスを行います．つまり，テーブルの内容に従ってFATやディレクトリの位置を決定し，その該当するセクタへのアクセス命令をデバイス・ドライバに対して発行します．そのデバイスの種類が何であるかについては，まったく関知しません．ですから，ランダム・アクセスのできるセクタを持った構造のデバイスであれば，どのようなものでも利用できるということになります．

次にBPBの各エントリについて簡単に説明しておきます．

○1セクタのバイト数

これは，"MSDOS.SYS"がアクセスするときのセクタのサイズが何バイトであるかを決めるものです．これは実際のデバイスの物理セクタサイズと一致させる必要はありません．たとえば，PC-9800シリーズのMS-DOSでは，ハードディスクのセクタサイズを1Kバイトとしています．しかし，実際の物理的なセクタサイズは256バイトですから，4セクタを1つのセクタとみなしてI/Oリクエストが行われているわけです．なお，このセクタサイズは，少なくとも32の倍数になっていなければなりません．

○1アロケーション単位あたりのセクタ数

アロケーション単位とは，別名クラスタとも呼ばれるファイルへの割当ての単位のことを指します．詳しくは5章を参照してください．このフィールドは，2のべき乗(1を含む)である必要があります．

○予備のセクタ

これは，ディスクの先頭から何セクタをMS-DOSの管理下から除外するかを決めます．MS-DOSの標準ディスクフォーマットでは，この数は1となっており，通常はブート用のプログラムを置くために使われています．つまり，FATはトラック0のセクタ2(ディスクの2番目のセクタ)からはじまるということです．RAMディスクのように，ブートの手続きが不用なものは0に

しておくとよいでしょう．また，1つのハードディスクを他のDOS(CP／MやDISK BASIC)と共有しているようなハードウェア・システムでは，他のDOSが管理している部分を使わないように，この値が決められています(Fig. 4.11)．



Fig. 4.11　予備のセクタ

○FATの数

1つのディスクに含まれているFATの数を表示します(5章参照)．

○ディレクトリ数

ルート・ディレクトリ中のディレクトリ・エントリの総数(ディレクトリの占めるセクタの数ではなく，エントリの数)を決めます．

○論理イメージのセクタ数

そのディスクに含まれるセクタの総数を表します．これは，予備のセクタの数も含めた値です．

○メディア・ディスクリプタバイト

これは00H～FFHの値をとる，BPBの識別用データのためのものです．詳しくはのちほど述べます．

○FATセクタ数

1つのFATが何セクタを占めるかを表したものです．

4.3.2　BPBの選択

さて，今説明したBPBを選択して〝MSDOS．SYS〟へ渡すのは〝IO．SYS〟の仕事です．〝MSDOS．SYS〟から〝IO．SYS〟へ，現在のドライブの中にはいっているディスクに関するBPBを渡すように，I／Oリクエストが行われます．さきほど，〝IO．SYS〟はディスクの内容については関与せずにI／Oリクエストの処理をすると述べましたが，BPBの選択に関しては〝IO．SYS〟にすべて任されているのです．つまり，〝IO．SYS〟はディスクのタイプの情報をディスクから読み取り，その結果，正しいBPBを〝MSDOS．SYS〟へ渡すわけです．

MS-DOS標準フォーマットのディスクでは，非常に簡単にディスクのタイプを判定すること

ができるようになっています．実は，それらのディスクでは，FATがかならずディスクの2番目のセクタからはじまっており，そのFATのいちばん先頭のバイトにFAT IDバイトと呼ばれるものがあります．そこにディスクのタイプが書かれているので，〝IO．SYS〟では，そのFAT IDバイトを読むだけで，ディスクタイプの識別ができます．その後，FAT IDバイトに応じて(5インチでは8種類，8インチでは3種類の)BPB群の中から正しいBPBを1つ選択して，それを〝MSDOS．SYS〟へ渡します(Fig. 4.12)．

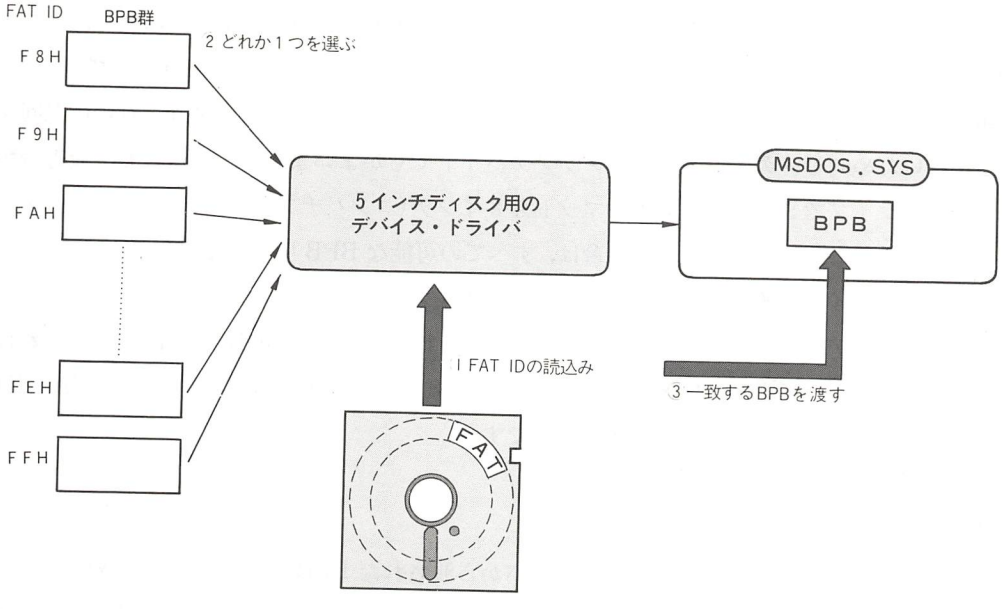


Fig. 4．12　BPBの選択

なお，このFAT IDバイトの値やそれに対応するBPBの構成が決まっているのは，8インチと5インチのディスクだけで，その他のものについては今のところ規格がありません．規格を作るほど，この2種類のほかに一般に広まっているデバイスがまだないからです．ですから，ユーザーがこの2種類以外のデバイスを使う場合，メディアタイプの識別にFAT IDを使用するか否かは自由であり，ほかにもっとよい方法があればそれを用いてもかまいません．要するに，正しいBPBが選択できれば，どのような方法でもかまわないのです．FAT IDによる識別は単に1つの手段にすぎません．ただし，固定式のハードディスクやRAMディスクなどのようにメディアが交換できないものでは，BPBの選択に迷う必要がないので，つねに1つのBPBを用意しておくだけで済みます．

4.3.3　メディア・ディスクリプタバイト

1つのドライブで何種類ものディスクをサポートする場合，何種類ものBPBが存在することはすでに述べたとおりです．このとき，数あるBPBのうち，現在〝MSDOS．SYS〟が使っているのはどのBPBであるかを簡単に知るために，BPBの中にメディア・ディスクリプタバイトと呼ばれる1バイトのある値をセットしておきます．もちろんセットするのは〝IO．SYS〟であり，異なったBPBには異なったメディア・ディスクリプタバイトを設定しておきます．この値は00H～FFHの任意の値をとることができますが，異なったタイプのBPBには異なった値を割り当てなければなりません．〝MSDOS．SYS〟にとって，値そのものはあまり重要ではなく，異なったBPBに対して異なった値が割り当てられているという事実だけが重要なのです．ただし，同一の種類のデバイスでない，つまり5インチと8インチのディスクのような場合は，互いに混同することがないため同じメディア・ディスクリプタバイトでもかまいません．もっとも，最近登場した1台のドライブで5インチのフォーマットも8インチ(コンパチブルな)フォーマットもアクセスできるようなドライブを使用した場合は，すべての可能なBPBについて異なったメディア・ディスクリプタバイトの値を設定する必要があります．

実際にメディア・ディスクリプタバイトの値を設定する場合には，00H～FFHの値であればどのような値を設定してもかまいませんが，5インチと8インチのディスクに対しては，FAT IDバイトと同一のものが使われることが多いようです．

4.3.4　DPB

〝MSDOS．SYS〟では，デバイス・ドライバから渡されたBPBを，ドライブの数(デバイス・ドライバの数ではない)だけ集めて，Fig. 4.13 のようなDPB(Drive Parameter Block)と呼ばれる〝MSDOS．SYS〟の内部の作業領域を作ります．

〝MSDOS．SYS〟はDPBを作成するために，各デバイス・ドライバに対して，〝BUILD BPB〟(BPBの作成)というI／Oリクエストを発行します．このI／Oリクエストにより得られたBPB(詳細は各リクエストの説明を参照)をDPBへコピーしておき，それ以後のファイル・アクセスは，このDPBを参照して行います．

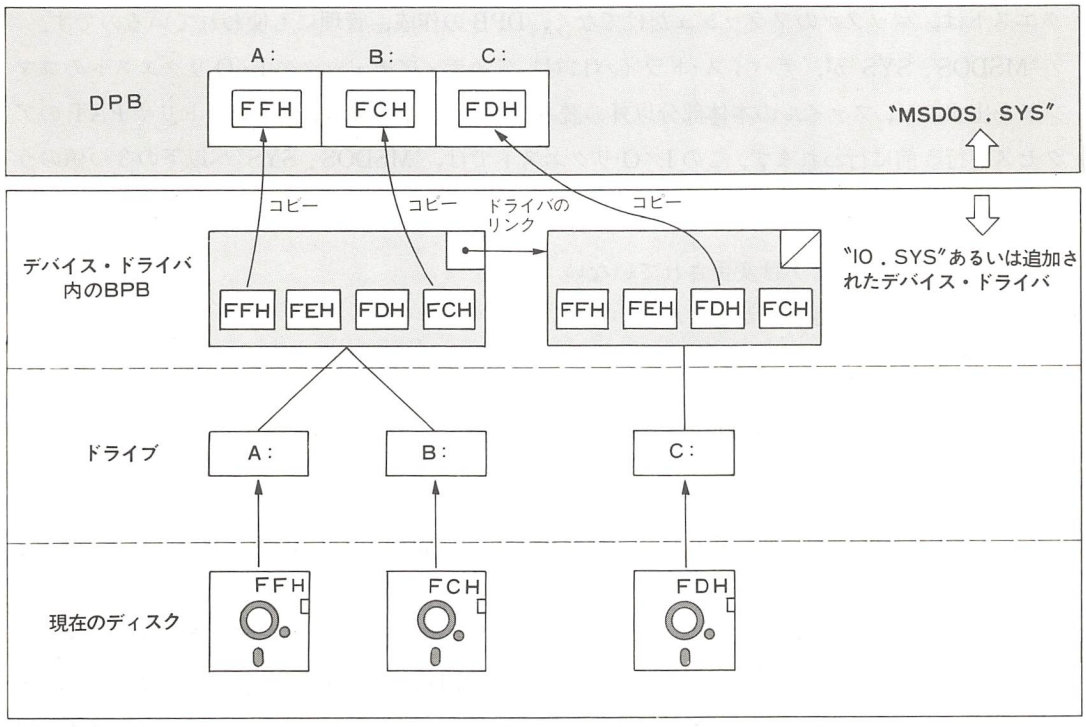


Fig. 4 .13　DPB(Drive Parameter Block)

4.3.5　メディアチェックとＤＰＢの更新

MS-DOS では，１つの種類のドライブで，幾種類かのディスクをサポートしなければならないため，ドライブと BPB の結び付きは固定的であってはなりません．ドライブの中にはいっているディスクに対応する BPB と DPB の内容はつねに一致させておかないと正常なアクセスが行われないため，実際のディスクのタイプを調べて，BPB を更新するかどうかを決めるということが必要になります．このことを〝メディアチェック〞と呼びます．〝MSDOS．SYS〞はデバイス・ドライバに対して，このメディアチェックのＩ／Ｏリクエストを発行してメディアが交換されたか否かを調べ，もし交換されていれば，現在の DPB を更新するためにデバイス・ドライバへ〝BPB の作成〞のＩ／Ｏリクエストを発行します．しかし，メディアが交換されていなければ現在の DPB の内容を正しい BPB であるとみなして，以後のディスク・アクセスを継続します．

さて，このメディアチェックをいつ行うかについてですが，実はもうすでに説明が済んでいるのです．3.8 でファイル・アクセスとディスク・バッファリングについて解説したとき，バッファの内容が正しいか否かを調べるために〝IO．SYS〞へディスクが交換されたかどうかを調べるコマンドを発行するという話をしました．そしてその結果によって，バッファをすべて捨てるかそのまま使うかを決定していたわけですが，その〝IO．SYS〞へコマンドを発行するという動作は，メディアチェックのＩ／Ｏリクエストのことだったのです．つまり，メディアチェックのＩ／Ｏリ

クエストは，バッファのフラッシュだけでなく，DPBの作成，管理にも使われているのです．

〝MSDOS．SYS〞が，デバイス・ドライバに対してメディアチェックのI／Oリクエストのコマンドを出すのは，ファイルの本体部分以外の読み書きのとき，つまり，ディレクトリやFATのアクセスを行う前に行われます．このI／Oリクエストでは，〝MSDOS．SYS〞へ以下の3つ値のうちのいずれかを返します．

(1)　1……メディアは変更されていない
(2)　0……変更されたかどうかわからない
(3)－1……メディアは変更された

これらの値に対する〝MSDOS．SYS〞の対応は次のとおりです．

(1)メディアは変更されていない
　この場合は，現在のDPBは正しいものとして継続してディスク・アクセスを行います．
(2)変更されたかどうかわからない
　このときの対応は，そのドライブに関するダーティバッファの有無によります．ダーティバッファが存在する場合，そのドライブのDPBは正しいものとして以降のディスク・アクセスが行われます((1)と同じ)．ダーティバッファがない場合，ディスクは変更されたものとみなされて，次に述べる(3)と同じ処理が行われます．
(3)メディアは変更された
　この場合，そのドライブに関するDPBは捨てられ，新たにDPBを作成するために，デバイス・ドライバに対してBPBの作成を要請します．そして，これにより得られたBPBをDPB内に組み込みます．

4.4　I／Oコマンド

MS-DOSにおけるI／Oの概略については，すでに述べてあるので，ここでは各I／Oリクエストについて詳しく述べていくことにします．

まず4.4.1で，各I／Oリクエストの際に必要となるコマンドパケットの詳細に触れておきます．その後，4.4.2で各I／Oリクエストについて解説していきます．

4.4.1　リクエストヘッダ

○コマンドパケットとリクエストヘッダ

コマンドパケットは，コマンドの種類やコマンドパケットの長さなどの情報を入れた〝リクエ

ストヘッダ″と呼ばれる固定長部分と，コマンドにより長さが可変のパラメータ受渡し用の2つの領域から成っています．

リクエストヘッダは，Table4．3のように13バイトの大きさから成る領域のことであり，I／Oリクエストの種類(コマンド)によってこの直後からさらに数バイトの領域が続きます．

Table　4．3　リクエストヘッダ

1バイト	レコード長
1バイト	論理装置コード
1バイト	コマンドコード
1ワード	ステータス
8バイト	予約域

○レコード長

このコマンドパケットの長さをバイト単位で表したものです．これには，リクエストヘッダの長さと可変長部の長さの両方を足したものがはいっています．

○論理装置コード

ブロック型デバイスに対してのみ意味があります．このタイプのデバイス・ドライバでは，1つのもので複数のデバイス(ドライブ)がサポートできるため，I／Oリクエストがどのデバイスに対して発行されたかを識別するために，この論理装置コードが使用されます．たとえば，あるデバイス・ドライバが3台のディスクをサポートしていたとすると，このフィールドの値は0～2のいずれかになります．1台のデバイスしかサポートしないデバイス・ドライバでは，このフィールドはつねに0となり，また意味もないので無視してもかまいません．

○コマンドコード

次の項で詳しく述べますが，ここにはI／Oリクエストの種類を表すコードがはいります．コードは0から12のいずれかの値をとります．

○ステータス

I／O処理の結果を表すためのステータスを置くための領域です．詳細は後述します．

○DOSのための予約域(8バイト)

この領域は現在使われておらず00Hがはいっています．しかし4.2で述べたように，将来MS-DOSがマルチタスク版になったとき，この領域は2つのDWORDのポインタのために使用されるでしょう．1つは"MSDOS．SYS"が各コマンドパケットの位置を記憶しておくためのキュー

のリンク用であり，もう1つは各デバイス・ドライバに対するI／Oリクエストを入れておくためのキューのリンク用になると思われます．もちろんI／Oリクエストのキューのリンクを管理するのは各デバイス・ドライバのストラテジルーチンになるでしょう．

4.4.2　ステータスとI／Oリクエスト

コマンドパケット中のステータスワードは，最初は0000Hになっています．そして，デバイス・ドライバは，I／O処理が終わった後，このステータスワード中のdoneビットを1にセットしてから戻らなければなりません(Fig.4.14)．

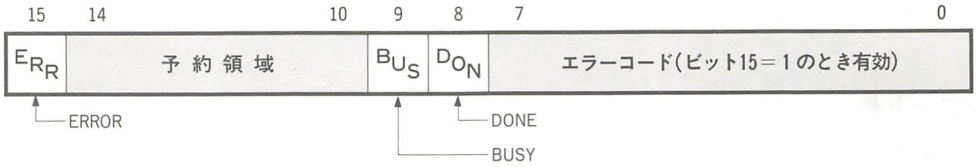


Fig. 4 .14　ステータスワード

ステータスワード中のビット15はエラービットであり，I／O処理中にエラーが生じたら，このビットとdoneビットを1にセットして，ステータスワードの下位バイトにエラー情報をセットして戻ります．エラー情報はTable 4.4のものを返さなくてはなりません．

Table　4 . 4　エラー情報

番号	エラー内容
0	ライトプロテクトされているディスクへの書込み
1	存在しないユニット番号
2	ドライブの準備ができていない
3	コマンド番号が間違っている
4	CRCエラー
5	コマンドパケットの長さが間違っている
6	シークエラー
7	メディアタイプが間違っている
8	セクタが見つからない
9	プリンタの用紙切れ
10	書込みエラー
11	読出しエラー
12	その他のエラー

ビット9(busy)は，I／Oリクエスト6または10で0または1にセットします．そのほかのI／Oリクエストでは関係ありません．詳しくは後述します．

4.4.3 I／O リクエストコマンド

デバイス・ドライバで処理しなければならないコマンドは，Table 4.5 に示すように全部で 13 個あります．分類すると，初期化，BPB の作成，リード（デバイス／IOCTL），ライト（デバイス／IOCTL），ステータスチェックの 5 種類になります．以下ではこれらのコマンドをこの分類に沿って解説していくことにします．

Table 4．5　I／Oリクエストコマンド

コマンドコード	ファンクション
0	INIT
1	MEDIA CHECK（ブロック型のデバイス・ドライバのみ，キャラクタデバイスの場合NOP）
2	BUILD BPB（ブロック型のデバイス・ドライバのみ，キャラクタデバイスの場合NOP）
3	IOCTL INPUT（デバイス・ドライバがIOCTL機能を持つ場合のみコールされる）
4	INPUT（リード）
5	NON-DESTRUCTIVE INPUT NO WAIT（キャラクタデバイスのみ）
6	INPUT STATUS（キャラクタデバイスのみ）
7	INPUT FLUSH（キャラクタデバイスのみ）
8	OUTPUT（ライト）
9	OUTPUT（ライトとベリファイが行われる）
10	OUTPUT STATUS（キャラクタデバイスのみ）
11	OUTPUT FLUSH（キャラクタデバイスのみ）
12	IOCTL OUTPUT（デバイス・ドライバがIOCTL機能を持つ場合のみコールされる）

○ INIT

コマンドコード＝0

13バイト	リクエストヘッダ	[R／W]
1バイト	ユニット数	[W]
2ワード	ブレーク・アドレス	[W]
2ワード	ＢＰＢの配列へのポインタ （キャラクタ型デバイスではセットしない）	[W]

注）　[R]……デバイス・ドライバ側では参照のみ許される
[W]……デバイス・ドライバ側では書き込む（セットする）
[R／W]……デバイス・ドライバ側で参照と書き込みをおこなう

このコマンドは，デバイス・ドライバが設置されたときに一度だけ実行されます．つまり，システムが起動したとき，各デバイス・ドライバの初期化をする目的のためのコマンドです．このコマンドを受けたら，デバイス・ドライバのルーチンの初期化を行い，以下で述べる 3 つのフィールドをそれぞれ正しい値にセットしてから戻らなければなりません．

ユニット数は，このデバイスがブロック型であるときに，このデバイス・ドライバで何台のドライブをサポートしているかを表した数です(1から始まる)．キャラクタ型デバイスでは1にしておくとよいでしょう．

ブレーク・アドレスはデバイス・ドライバがどこで終わっているか，つまり，メモリ上のどこまでを占めているかを表すアドレスです．〝MSDOS．SYS〟は，このアドレス以降を空き領域であるとみなして，ユーザープログラムや次のデバイス・ドライバをロードします．ですから，1回限りで不要になるINITルーチンをデバイス・ドライバの最後部へ置き，ブレーク・アドレスをそのルーチンのスタート・アドレスにセットしておけば，メモリを節約することができます．

BPB配列に対するポインタは，各ドライブに対する最初のBPBの状態を設定するためのものです．〝MSDOS．SYS〟は各ブロック型のデバイス・ドライバからBPBを集め，DPBを作ります．ここで渡すBPBは，仮のものでもよいのですが(後で正しいBPBを返す)，メディア・ディスクリプタバイトとセクタサイズには注意が必要です．もし仮の(正しくない)BPBを返すのであれば，メディア・ディスクリプタバイトも存在しない値を入れておかなければなりません．またセクタサイズは，デフォルトの〝IO．SYS〟組込みのどのデバイス・ドライバのものよりも大きくすることはできません．もしこれを許すと，デフォルトのディスクバッファ(2つ)の大きさまで変更しなければならなくなるからです(このバッファは，デバイス・ドライバのロード時にはもうすでにメモリ中に確保されている)．

このフィールドへ入れるのは，BPBへのポインタをユニット数だけ集めた配列へのポインタです．〝MSDOS．SYS〟は，このポインタ群からBPBをシステム領域へ(各BPBの内容を)コピーしてDPBを作るので，同じBPBを返すつもりなら，BPBの実体は1つだけ用意しておいて，2つ以上から1つのBPBを指すようにしてもかまいません．

以上3つのフィールドのことを図で表すとFig. 4.15のようになります．

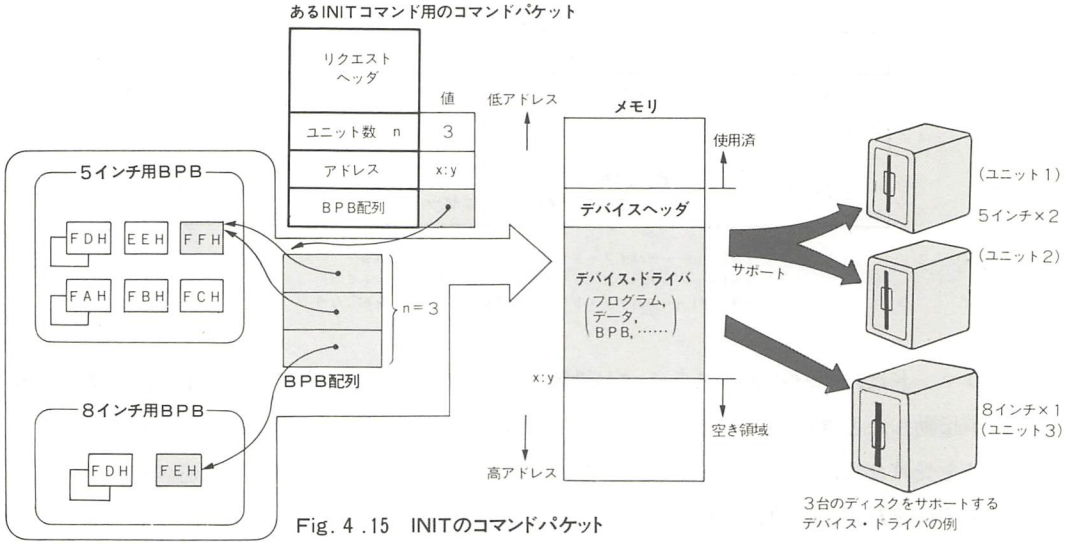


Fig. 4.15　INITのコマンドパケット

なお，〝CONFIG．SYS〟で登録したデバイス・ドライバの場合，コマンドパケットの最後の2ワード(BPB配列へのポインタ)には，〝CONFIG．SYS〟中の〝＝〟の次の文字へのポインタがはいっています．ですからこのポインタを使うと，デバイス・ドライバを起動する場合にパラメータを与えることができます(Fig. 4.16)．たとえば，RAMディスクのドライバを作成した場合に，この文字列で全容量を与えるようにすると，いちいちデバイス・ドライバをアセンブル(やコンパイル)しなくても，エディタで〝CONFIG．SYS〟を編集し直すだけで簡単に容量の変更ができます．

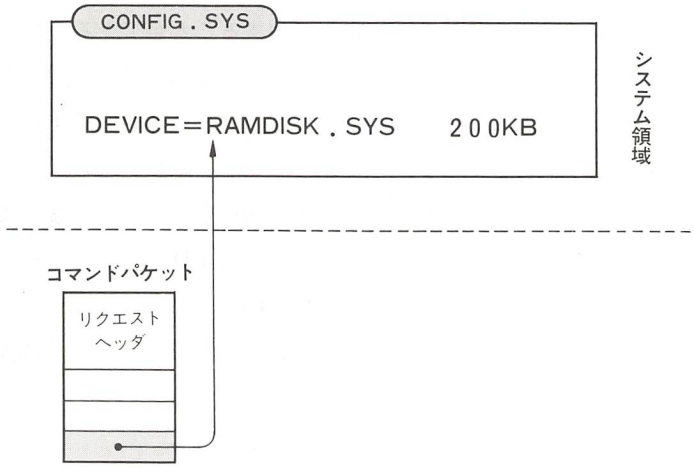


Fig. 4．16　文字列パラメータ

○ MEDIA CHECK

コマンドコード＝1

13バイト	リクエストヘッダ	[R／W]
1バイト	DPBからのメディア・ディスクリプタ	[R]
1ワード	返す値	[W]

返す値はすでに述べたように，以下の3つのうちのいずれかです．

−1　メディアは変更されている
0　変更されたかどうかわからない
1　メディアは変更されていない

メディアを交換することのできるデバイスにおいて，交換されたかどうかを調べるためには，やはり何らかのハードウェアの支援があった方が，判定するための手間が少なくて済みます．

いちばん簡単な方法は，ドアロックを用いる方法でしょう．ディスクがセットされて最初にアクセスされたときシステム側からドアロックをしてしまい(多くの8インチディスクでは，オプシ

ョンでドアロック機構を付けられる),以後デバイス・ドライバは1を返し続けるというものです.しかしこれではディスクの交換が行えないので,簡単にドアロックを解除するための手段が必要です.また,CPUが暴走しているときでも解除できるようにしておかないと,ディスクが完全に破壊されるというようなことが起こる可能性があります.

これを少し改良した方法として,ディスクへのアクセス後数秒～数十秒の間だけドアロックをしておき,この間は1を返すという方法があります.この方法を用いた場合,ドアロックと解除のための音が騒々しいという欠点があります.

この方法は,他のドアロックのないシステムにも応用することができるでしょう.たとえばラストアクセスから,2秒間だけは1を返すようにするという方法があります.この2秒という時間は,とてもその間にはディスク交換ができないであろうと思われる時間ですが,普通に使用している限りこの方法だけでも,随分MS-DOSの反応が速くなります.

しかし,メディアが交換されたことを知る手段で最も簡単でエレガントなのは,やはり何らかのハードウェアによる支援を行った方法でしょう.たとえば,FDC(フロッピーディスクコントローラ)の1つであるNECのμPD765では,自動的にこのチェックを行います.このようなFDCを使うのも1つの方法でしょう(ただし,この場合使用できるドライブにやや制限がある).

○ BUILD BPB

コマンドコード＝2

13バイト	リクエストヘッダ	[R／W]
1バイト	ＤＰＢへのメディア・ディスクリプタ	[W]
2ワード	転送アドレス	[R]
2ワード	ＢＰＢに対するポインタ	[W]

これは,BPBの作成をするためのI／Oリクエストです.デバイス・ドライバ側では,現在ドライブにはいっているディスクのタイプを調べてそれに合うBPBを作成(または選択)し,そのBPBへのポインタ(DWORD)をコマンドパケット内のBPBに対するポインタフィールドへセットしなければなりません(INITコマンドと違って,BPBの配列へのポインタではなく,BPBそのものへのDWORDポインタを返さなければならない).

さて,ディスクのタイプのチェックについてですが,もしディスクがMS-DOS標準フォーマットならば,ディスクの2番目のセクタ(FATのはじまりのセクタ,トラック0,セクタ1,サイド0)を読み出し,その先頭バイト(FAT IDバイト)の値に従って該当するBPBを選択すればよいのです.この作業はこれ以外には方法がないため,MS-DOS標準ディスクフォーマットしか扱わないデバイス・ドライバ(デバイスヘッダのアトリビュートワードのビット13が0のデバイス)に対しては,"MSDOS.SYS"側でFATの第1セクタを読み出した状態でI／Oリクエストのコー

ルが発行されます．そして，その読み出したバッファの先頭アドレスはコマンドパケットの転送アドレスフィールドにセットされています．ですから，デバイス・ドライバ側で行う処理は，単にそのバッファの1バイト目を読んできて，その値(FAT ID)に応じてBPBを選択し，そのアドレスをコマンドパケット内へセットするだけです．

これに対し，アトリビュートワードのビット13がセットされているデバイス・ドライバでは，デバイス・ドライバだけでBPBを作成しなければなりません．このとき，コマンドパケット内の転送アドレスフィールドには，自由に使用してよい1セクタ分のスクラッチ領域のスタート・アドレスがはいっているので，メディアタイプの決定に必要なら使ってもかまいません．もっとも，このデバイス・ドライバがたった1つのタイプのメディアしかサポートしない(できない)のであれば，めんどうな判別手続きを経なくても，即正しい値を返すことができます．

メディア・ディスクリプタバイトのフィールドには，作成したBPBに対応するメディア・ディスクリプタバイトの値(BPB内のものと同一のもの)をセットします．

なお，このI／Oリクエストにより得られたBPBのメディア・ディスクリプタバイトの値が，現在のDPB内のBPBのそれと一致する場合は，DPBの更新は行われません．ですからかならず，1ユニットで扱うことのできる異なるBPBに対して，すべて異なる値を割り当ててください．さもないと，誤ったディスク・アクセスが行われる可能性があります．

○リードまたはライト

コマンドコード＝3，4，8，9，12

13バイト	リクエストヘッダ	[R／W]
1バイト	ＤＰＢからのメディア・ディスクリプタ	[R]
2ワード	転送アドレス	[R]
1ワード	バイト／セクタ・カウント	[R／W]
1ワード	開始セクタ番号 (キャラクタ型デバイスではセットしない)	[R]

このI／Oリクエストは，ディスクやキャラクタ・デバイス，そしてデバイス・ドライバ自体とデータをやり取りするためにあります．デバイス・ドライバのアクセスとは，すでに何回か述べてきたIOCTLのことを指します．このIOCTLとディスクのアクセスは，目的こそ違いますが使用するコマンドパケットの形式はまったく同じで，転送するデータの数え方，つまり単位が異なっています．一方はセクタがもう一方はバイトが単位となっています．

次にコマンドパケット内の各フィールドの内容について説明しておきます．

メディア・ディスクリプタバイトは，現在のドライブの中にはいっているディスク用のBPB内のものですので，これがわかればもとのBPBの内容がわかるはずです．なぜなら，そのBPBは

デバイス・ドライバの方から渡したものだからです．このBPBを使ってディスクのタイプを知り，そしてディスクにアクセスします．

転送アドレスは，データの転送を開始するアドレスを保持しています．

バイト／セクタ・カウントは，読み書きするIOCTL文字列の長さ(I／Oリクエスト3，12)または転送するセクタの数(I／Oリクエスト4，8，9)を表します．

開始セクタ番号は，データの転送を開始する論理セクタ番号のことを指します．論理セクタ番号とは，ディスクに含まれるセクタに対して先頭から順に付けた番号のことで，0がトラック0，サイド0，セクタ1に相当します．なお，当然ながらキャラクタ型デバイスに対してはこのフィールドは意味がありません．

このコマンドでは，I／O処理が終わった後，何バイトまたは何セクタのデータを転送したかをコマンドパケット内のバイト／セクタ・カウントのフィールドへ正しくセットしなければなりません．つまり，要求されたI／O処理が正しく行われたならば，そのままFAR RETURNしてもかまいませんが(要求された量が転送された量と等しいので，もう一度セットする必要がない)，途中でエラーが発生した場合は，エラービットをセットするだけでなく，再試行(リトライ)をどう行うかを決めるために，データをどこまで転送したかを正しく"MSDOS.SYS"へ返さなければならないということです．ただし，IOCTL文字列の入出力ではエラーチェックは行われませんが，このデータの転送バイト／セクタを示すフィールドを正しくセットしなければなりません．

○NON DESTRUCTIVE INPUT, NO WAIT

コマンドコード＝5

13バイト	リクエストヘッダ	[R／W]
1バイト	デバイスからのデータ	[W]

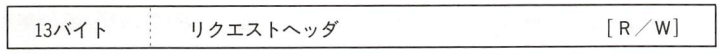
このコマンドは，キャラクタ・デバイスから先読みをするために用いられます．先読みとは，現在のキーボードのキュー(先行入力用バッファ)の先頭から1つ(1バイト)データを取ってくることを指します．しかし，このデータをキューから外してはなりません．つまり，キューを破壊せずに(NON DESTRUCTIVE)データを先読みするだけのコマンドです．

先読みしたデータはコマンドパケット内のデバイスからのリードというフィールドへセットします．

ただし，先読みするデータがない場合は(先行入力がない場合)，戻るときにリクエストヘッダ内のステータスワード中のdoneビットをセットするだけでなく，busyビットもセットしておかなければなりません．また，このとき，次のキー入力を待ったりせずに，すみやかに(NO WAIT)戻らねばなりません．

○STATUS

コマンドコード＝6，10



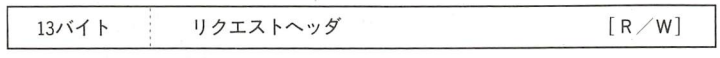
これらのコマンドは，現在のデバイスの状態を把握するために用いられます．結果は，ステータスワード中のbusyビットをセット／リセットすることによって返します．

コマンド6(INPUT STATUS)では，キー入力バッファが空ならbusyビットを1にセットし，バッファに何かはいっていればbusy＝0としておきます．つまり，キー入力において，バッファが空であれば実際の入力まで待たされるので，busyであるということです．

コマンド10(OUTPUT STATUS)は現在の出力デバイスの状態を調べるために使用されます．出力デバイスに対してすぐに文字を出力することができる場合busyビットは0とし，出力がすぐ行われず待つ必要があるときはbusy＝1としておきます．後者はプリンタヘッドの改行待ちとかRS-232Cのデータ出力待ちのようなときに発生します．

○FLUSH

コマンド＝7，11



これらのコマンドは，キャラクタ・デバイスで保持しているキューの内容をすべて捨てさせるためにあります．しかし実際には，コマンドコード＝7のみが使われ，キー入力バッファの内容を空にするためだけに使われます．

5章　ファイル・システム

ユーザーがDOSを扱う際には，木構造のディレクトリをアクセスしたり，ファイルを名前でオープンして入出力を行うことができますが，ファイルやディレクトリ自体がディスク上でどのように実現されているかは意識する必要はありません．このようにユーザーが〝論理的〟に扱うファイルやディレクトリといった概念を，ファイルを扱う単位やその区別を実際のディスク上に〝物理的〟に対応させ，ファイルやディレクトリ自体を構成する体系をファイル・システムと呼びます．

DOSにおけるファイル・システムの構成は，その処理速度や信頼性に大きく関係する要素です．ディスクに対する物理的なアクセスはセクタ単位で行われますが，ディスクの使用効率を上げるためにセクタサイズを小さくすれば，ディスクを頻繁にアクセスするために実行速度が低下しますし，あまり大きくすれば逆の結果となります．

また，ファイルの位置を検索する場合，その道筋が複雑であったり，ファイルがディスク上に点在していたとすれば，ファイルのアクセスが遅くなってしまいます．

このようにファイル・システムを構築する上では，最低限DOSの高速性とディスクの使用効率を考慮しなければならないとともに，そのファイル・システムはユーザーにとって使いやすいものとして設計される必要があります．本章では，MS-DOSのファイル・システムが，上で述べたことをどのように実現しているかを解説していきます．

5.1　ディレクトリとファイル

MS-DOSのファイル・システムは，UNIXとCP／Mをベースにしており，その最も大きな特徴は階層的ディレクトリ構造を実現していることでしょう．本節では，このディレクトリ構造がどのように実現され，ディレクトリとファイルがどのように扱われているのかについて見ていくことにします．

5.1.1　ディレクトリ・エントリとファイル

ディレクトリの中には，ファイル名やディレクトリ名と，それに付随する情報がはいっていますが，その個々の情報のことをディレクトリ・エントリと呼び，1つあたり32バイトの領域を占めています．Fig. 5.1は，実際のディスクのディレクトリ領域のダンプ例と各ディレクトリ・エントリのフィールドを示したものです．

```
A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19      ……普通のファイルの例
LOCAL       <DIR>      84-04-18  20:51      ……ディレクトリ
       2 個のファイルがあります
 1172480 バイトが使用可能です

A>copy command.com test↵
       1 個のファイルをコピーしました

A>del test↵

A>debug↵
-l0 0 5 1↵
-d0 1100↵
1B1E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00   IO      SYS'....
1B1E:0010  00 00 00 00 00 00 54 4C-5F 07 02 00 00 A0 00 00   ......TL_.... ..
1B1E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00   MSDOS   SYS'....
1B1E:0030  00 00 00 00 00 00 40 72-33 07 2A 00 00 45 00 00   ......@r3.*..E..
1B1E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00   COMMAND COM ....
1B1E:0050  00 00 00 00 00 00 6E 82-33 07 3C 00 26 43 00 00   ......n.3.<.&C..
1B1E:0060  4D 53 2D 44 4F 53 20 20-20 20 20 08 00 00 00 00   MS-DOS      ....
1B1E:0070  00 00 00 00 00 00 4F A6-92 08 00 00 00 00 00 00   ......O&........
1B1E:0080  4C 4F 43 41 4C 20 20 20-20 20 20 10 00 00 00 00   LOCAL       ....
1B1E:0090  00 00 00 00 00 00 77 A6-92 08 4D 00 00 00 00 00   ......w&..M.....
1B1E:00A0  E5 45 53 54 20 20 20 20-20 20 20 00 00 00 00 00   eEST        ....
1B1E:00B0  00 00 00 00 00 00 6E 82-33 07 4E 00 26 43 00 00   ......n.3.N.&C..
1B1E:00C0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   .eeeeeeeeeeeeeee
1B1E:00D0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   eeeeeeeeeeeeeeee
1B1E:00E0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   .eeeeeeeeeeeeeee
1B1E:00F0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5   eeeeeeeeeeeeeeee
```

消去されたディレクトリ・エントリを作る

ディレクトリ領域の開始セクタ番号

DEBUGを使って，ディスク中のディレクトリ・エントリを調べる．使用したディスクは8インチ両面倍密度タイプ

hidden file

未使用マーク

ディレクトリを示す属性

このようにたいていのファイルには，20Hの属性が付いている

未使用ディレクトリ・エントリ

消去されたディレクトリ・エントリ

ボリュームラベル

：ディレクトリの長さのフィールドはつねに0

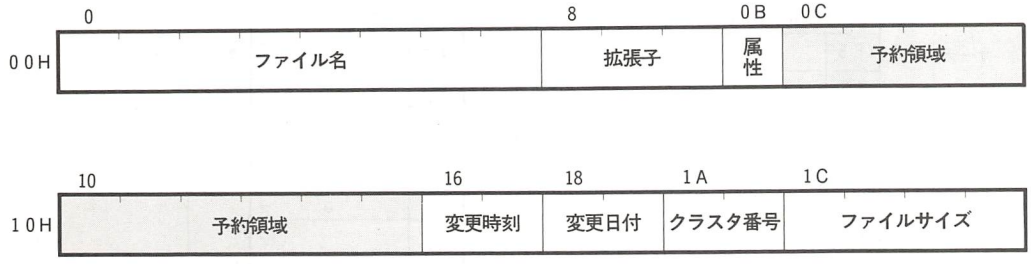


Fig. 5.1　ディレクトリ・エントリ

○ファイル名

ファイル名とファイル拡張子の欄は，MS-DOSで許されているすべてのファイル名(漢字でもよい)が左詰めではいり，余った部分には空白のコード(20H)を入れておきます．ここには，CONやPRNのようなデバイス名をDEBUGなどで(無理やり)置くこともできますが，そういうことをすると，そのファイルは参照できなくなってしまうでしょう．なぜなら，MS-DOSではファイルを捜すときに，まずデバイス名であるかどうかのチェックをするからです．

ファイルが削除された場合，ディレクトリ・エントリの先頭の1バイトにはE5Hを入れておき，このエントリが空いていることを記しておきます．これに対し，まだ一度も使われたことのないディレクトリ・エントリの先頭バイトには00Hがはいっています．フォーマットされたばかりのルート・ディレクトリや，作られてすぐのサブ・ディレクトリ中の未使用ディレクトリ・エントリの先頭には00Hが書き込まれているのです．この00Hは，ファイルの検索において(ファイルの検索は，ファイルのオープンや削除のときをはじめ，かなり頻繁に行われる)，ディレクトリ領域のどこまで捜せばよいかを知るために用いられます．Fig. 5.2に示すように，ファイルの検索において00Hがないと全ディレクトリ領域が調べられますが，途中に00Hがあった場合は，そこで検索が打ち切られるためにファイルの検索に要する時間が短くて済みます．ただし，これはファイルが存在しなかった場合の話であり，目的のファイルが存在している場合は，どちらでも検索時間に差がないことはいうまでもありません．

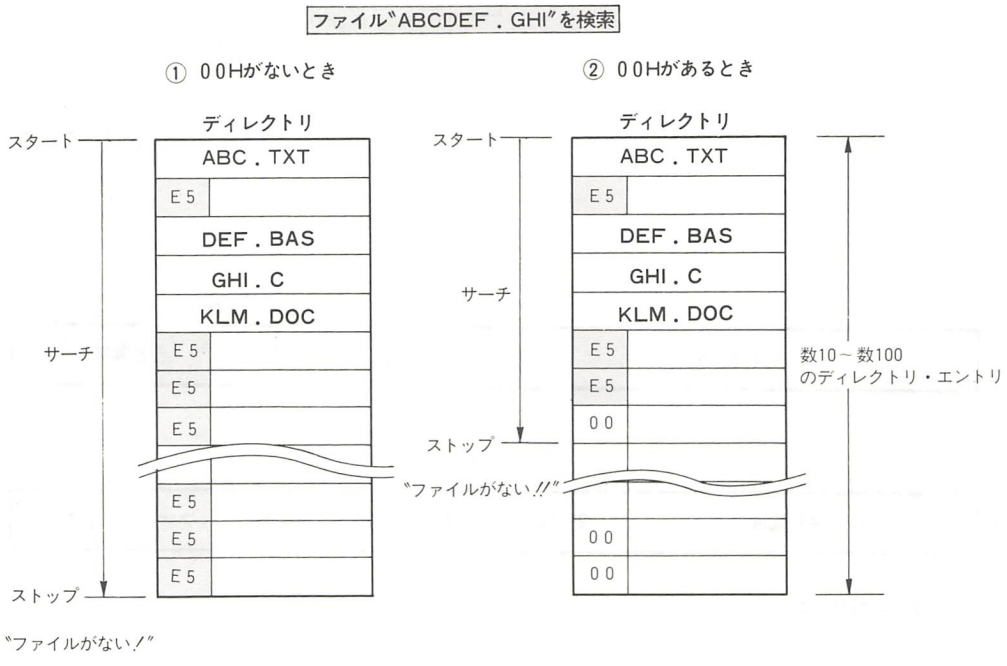


Fig. 5．2　ファイルの検索

○ファイルの属性

ディレクトリ・エントリのオフセット0BHには，Table 5.1に示すファイル(またはディレクトリ)の属性を示す値がはいっており，それぞれの値は以下に示すような意味を持っています．

Table　5．1　ファイルの属性

属性	意　味
01H	読出し専用ファイル
02H	不可視属性．通常のディレクトリサーチから除外される
04H	システムファイル
08H	ボリュームラベル
10H	ディレクトリ・エントリ
20H	保存ビット

01H…ファイルが〝読出し専用〟であることを示し，書込みを行うとエラーとなります．

02H…ファイルを不可視状態にするためのビットです．このビット(ビット1)が1にセットされているファイルはDIRコマンドでは表示されなくなり，基本FCBを用いたシステムコールでは，〝File not found〟(ファイルが見つかりません)エラーとなります．もしこのファイルをアクセスする必要があるのであれば，拡張FCBもしくはXENIXコンパチブルなシステムコールを用いなければなりません．

04H…ファイルがシステムファイルであることを示します．このビット(ビット2)が1にセットされていると，02Hのビットと同様にDIRコマンドでは表示されなくなり，通常のシステムコールではアクセスできなくなります．〝IO．SYS〟，〝MSDOS．SYS〟などのシステム側にとって意味のあるファイルには，属性07Hが付いており(下位3ビットがセットされている)，ユーザーの書込みや削除から保護されるようになっています．

08H…ディレクトリ・エントリの名前欄と拡張子欄の計11バイトには，そのディスクのボリュームラベルがはいっています．このディレクトリ・エントリは，FORMATコマンドによってルート・ディレクトリにセットされますが，このとき，FORMATを実行した時刻も同時にディレクトリ・エントリの日付／時刻欄に記録されます．

10H…ディレクトリ・エントリがサブ・ディレクトリ名を表しているものと解されます．このディレクトリ・エントリの日付／時刻欄には，このサブ・ディレクトリを作成した時刻が記録されます．

20H…ファイルに対して書込みが行われたときに1にセットされます．ですから通常のファイルはすべてこのビットが1にセットされています．

以上述べてきた属性は，08Hと10H以外はすべて自由に組み合せて使うことができます．また，ファンクション43Hのシステムコールによって変更することも可能です．

○最終変更時刻

ディレクトリ・エントリのオフセット 16H～19H には，そのファイルの最終変更時間などの情報がはいっており，その記録形式は Fig. 5.3 のようになっています．

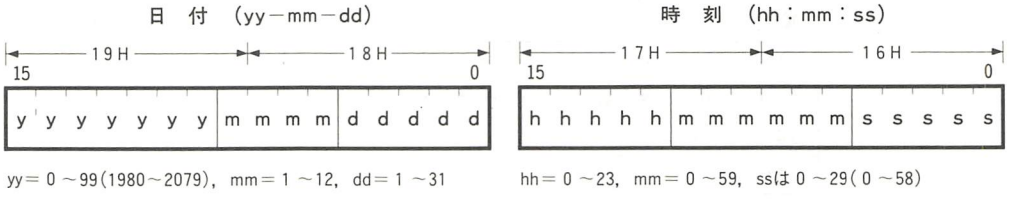


Fig. 5．3　日付／時刻のフォーマット

○クラスタ番号

オフセット 1AH～1BH には，後述するファイルやディレクトリの開始クラスタの番号がはいっています．クラスタとは，データの記録単位の 1 つですが，詳しいことは 5.2 を参照してください．

○ファイルサイズ

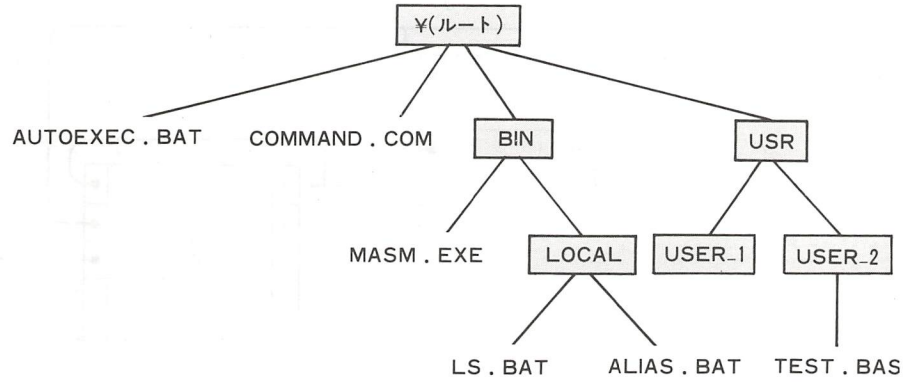
ディレクトリ・エントリの最後の 4 バイトには，そのファイルサイズをバイトで表したものがはいっています．4 バイトの値は，オフセット 1FH が最上位バイト，オフセット 1CH が最下位バイトとなっています．なお，このフィールドは，そのディレクトリ・エントリがボリュームラベルやサブ・ディレクトリ名を表しているときは意味がなく，単に 00H となっています．

5.1.2　ファイル・システムの実現

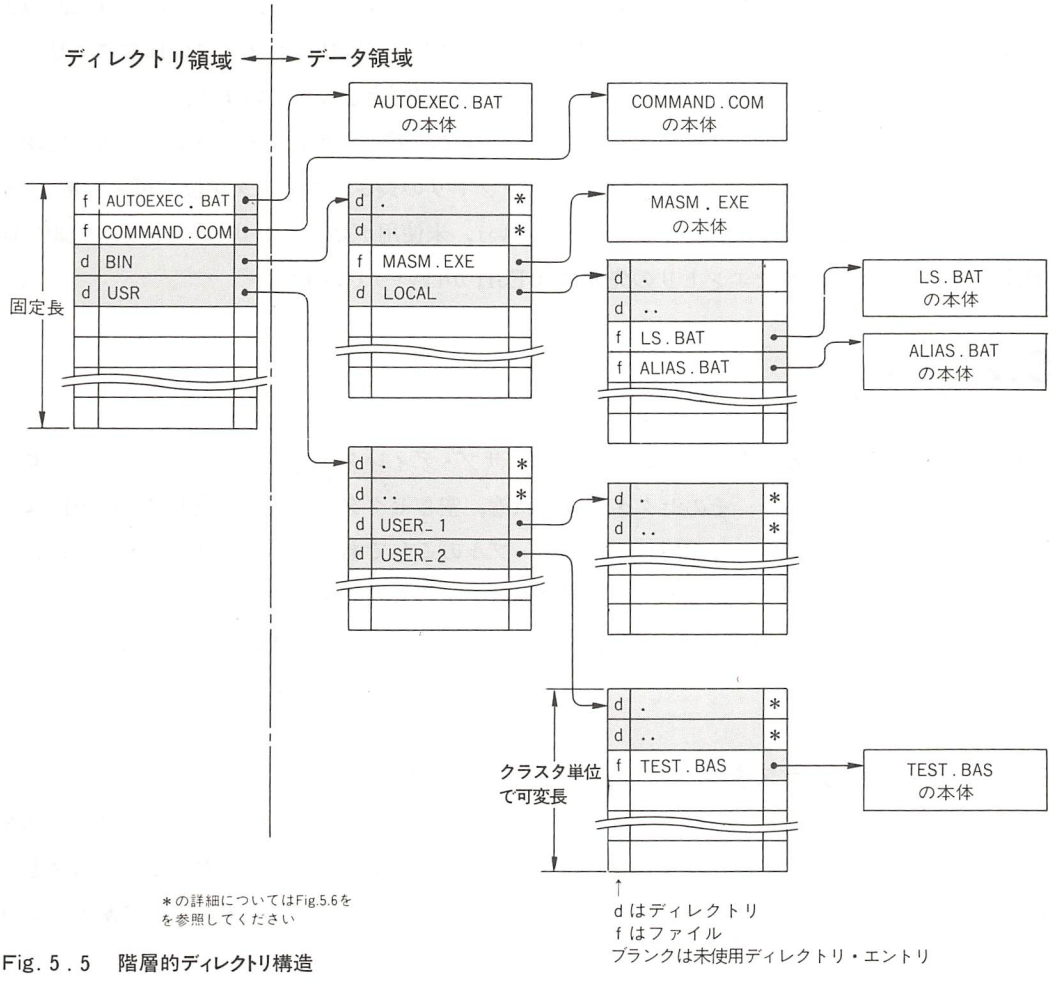
それでは，ディレクトリとディレクトリ・エントリが，実際のディスク上でどのようにしてファイル・システムを構成しているかを見ていきます．

ファイル・システムがディスク上でどのようにして実現されているかを理解してもらうために，Fig. 5.4 に示した構造が，ファイル・システムでどのように実現されているかを Fig. 5.5 に示しておきます．

Fig. 5.5 からわかるように，サブ・ディレクトリには，つねに〝.〟と〝..〟ディレクトリの 2 つが含まれています．図中では繁雑になるために省略してありますが，〝.〟ディレクトリは自分自身(サブ・ディレクトリ自身)へのポインタを書くべきであり，〝..〟ディレクトリのポインタ部分には，親ディレクトリへのポインタがはいっています．この部分を詳しく書くと Fig. 5.6 のようになります(ただし，一部のみ)．



Fig．5．4　階層的ディレクトリの例



Fig．5．5　階層的ディレクトリ構造

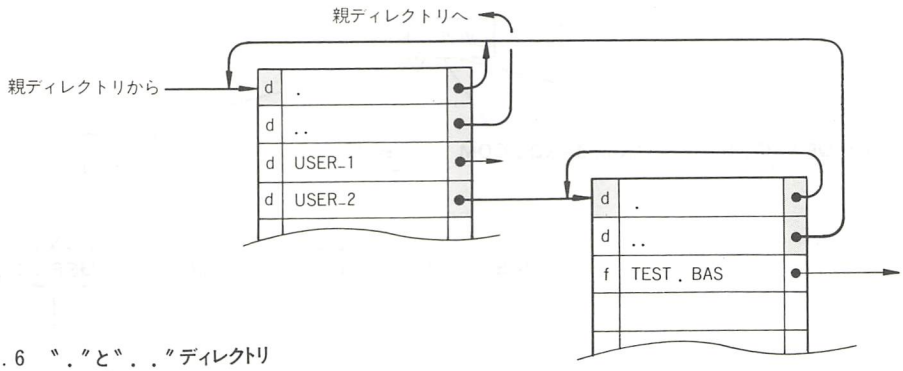


Fig. 5 . 6　〝.〟と〝..〟ディレクトリ

ルート・ディレクトリとサブ・ディレクトリの差は，〝.〟と〝..〟の有無だけではなくディレクトリ・エントリの最大数にも違いがあります．MS-DOS ver. 2.0 のディスクでは階層的ディレクトリ構造をサポートしていますが，ver. 1.25 のディスクを考慮しているために，ルート・ディレクトリだけはどちらのものでも互換性があるように作られています．そのため，ver. 2.0 のディスクのルート・ディレクトリは，ver. 1.25 のディレクトリと同様，ディスクの種類によりディレクトリ領域の大きさ，最大ディレクトリ・エントリ数が固定となっています．

これに対し，サブ・ディレクトリはファイルと同様に，ディレクトリ・エントリの数が可変となっています．ただし正確にいうと，サブ・ディレクトリの長さはクラスタ単位で可変であり(といっても大きくなる一方で，小さくなることはない)，未使用ディレクトリ・エントリの先頭には 00H がはいり，削除されたエントリの先頭には E5H がはいっています．

5.2 FAT

FAT(File Allocation Table)とは，ファイルやサブ・ディレクトリがディスク上のどこのセクタを使用しているか，また，そのセクタの連結状態，空きセクタの位置，不良セクタの位置などのディスクの使用状態について記憶しているテーブルのことであり，ディレクトリとともにディスク上のファイルを管理する上で非常に重要な働きをするものです．

本節では，まずセクタの関係について説明した後，そのクラスタと FAT について述べることにします．

5.2.1 セクタとクラスタ

MS-DOS では，ファイルのアクセスがセクタ単位で行われ，ディスクバッファリングもセクタを単位として行われていることは 3.8.3 で述べたとおりですが，ファイルに対してディスク上の物理セクタを割り当てる場合，セクタではなくクラスタと呼ばれるものを単位として割当てが行われます．

1クラスタは，物理セクタの2のべき乗倍となるように，ディスクの種類により決まっています．たとえば，5インチのフロッピーディスクでは1クラスタは1あるいは2セクタ(512～1024バイト)，8インチの両面倍密ディスクでは1クラスタは1セクタ(1024バイト)となっています(ハードディスクについては，メーカーに依存するようである)．

クラスタ単位でディスクを管理するのは，セクタという物理的なもののサイズや最大セクタ数などを意識せずに，ファイルが構成できるようにするためです．つまり，1セクタのサイズは，128，256，512，1024バイトなどの種類がありますが，セクタサイズが変わるたびにファイルへのアクセス法を変えていたのでは，プログラムがたいへん複雑になるため，より統一的にディスクを扱うための単位としてクラスタを設定し，それによってディスクを管理しているのです．ただし，クラスタ単位でバッファリングをしていたのではメモリの使用効率が下がるため，ディスクとの実際の入出力はセクタ単位で行うというようにして，MS-DOSではきめ細かなディスク・アクセスを行っています(Fig. 5.7)．

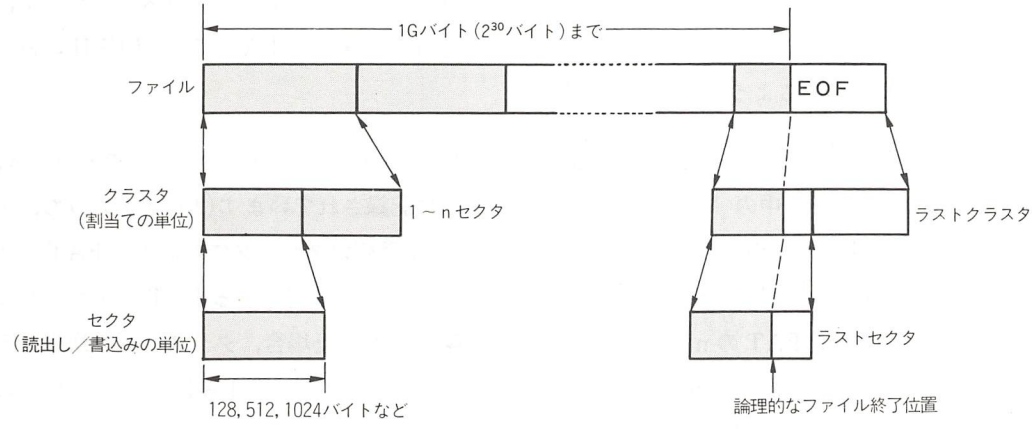


Fig. 5.7 ファイルとクラスタとセクタ

5.2.2 FATとクラスタ

ファイルはクラスタを単位として構成されていますが，クラスタのディスク上の物理的な位置はつねに連続しているわけではありません．そのため，FATによってそれらのクラスタがどのような順につながって，1つのファイルを構成しているかが管理されています．

FATの各エントリは，1つあたり12ビット(1.5バイト)，2つで3バイトの領域を占めます．この各エントリは，ディスクの各クラスタと1対1に対応しており，該当するクラスタの使用状況を表しています．クラスタの使用状況は000H～FFFHの値によって表され，それらの意味は次に示すようにいくつかに分類することができます．

000H…… 対応するクラスタは未使用状態であり，ファイルのデータ部やサブ・ディレクトリのために割り当てることができます．

001H…… このクラスタ番号が使用されることはありません．

002H
～FF6H… このエントリに対応するクラスタは，ファイルやサブ・ディレクトリの一部として使用中です．この番号は，対応するクラスタの次のクラスタ番号を表しています．

FF7H…… 対応するクラスタ中に，不良セクタがはいっています．不良セクタとは，なんらかの物理的要因(磁性体の欠陥やIDフィールドの欠陥など)により，読出しや書込みを行うことができないセクタのことを指します．クラスタに対してこの数値がセットされた場合，このクラスタは二度とファイルやサブ・ディレクトリの一部として使用されることはありません．

FF8H
～FFFH… 対応するクラスタは，ファイルやサブ・ディレクトリを構成するクラスタのうちで，いちばん最後のものであるということを示します．実際のFATでは，FFFHのみが使われているようです．

ファイルやサブ・ディレクトリを構成するクラスタのうち，最初のクラスタがどれであるかは，ディレクトリ・エントリ中の〝ファイル開始クラスタ〟に記録されています(2バイトのうち，下位12ビットが有効)．その次のクラスタがどこになるかは，最初のクラスタに対応するFATのエントリ中に書かれ，3番目のクラスタの番号は2番目のクラスタに対応するFATのエントリに書かれます．つまり，FATのn番目のエントリの内容がmだった場合，クラスタnの次はクラスタmが使われているというわけです．そして，最終クラスタに対応するFATのエントリには，FF8H～FFFHのどれかの値がはいっています(Fig. 5.8)．

FATのエントリにFF7Hがはいっていた場合，そのクラスタには不良セクタが含まれているため使用不可能であると解釈され，ファイルやディレクトリのために使われることはありません．

この印は，ディスクを最初にフォーマットするとき，FORMATコマンドによって記録されます．ディスク中に含まれる初期不良のセクタはこれにより検出されますが，使用中に生じた不良セクタはRECOVERコマンドにより検出除去されます．このコマンドは，指定されたファイルを先頭から読んでいき，読取りエラーの生じたクラスタを不良クラスタとみなしてFATのチェインから取り除き，FATのエントリをFF7Hに変更します．つまりこのコマンドの動作は修復ではなく，単にファイルとして使用されている不良クラスタを切り離しているだけなので，ファイルに割り当てられていないクラスタのチェックは行われず，たとえ未使用クラスタ中に不良クラスタがあったとしても，〝MSDOS．SYS〟ではそのクラスタを使用してしまいます．

なお，メーカーによってはFORMATコマンドで不良セクタの検出をしないものがありま

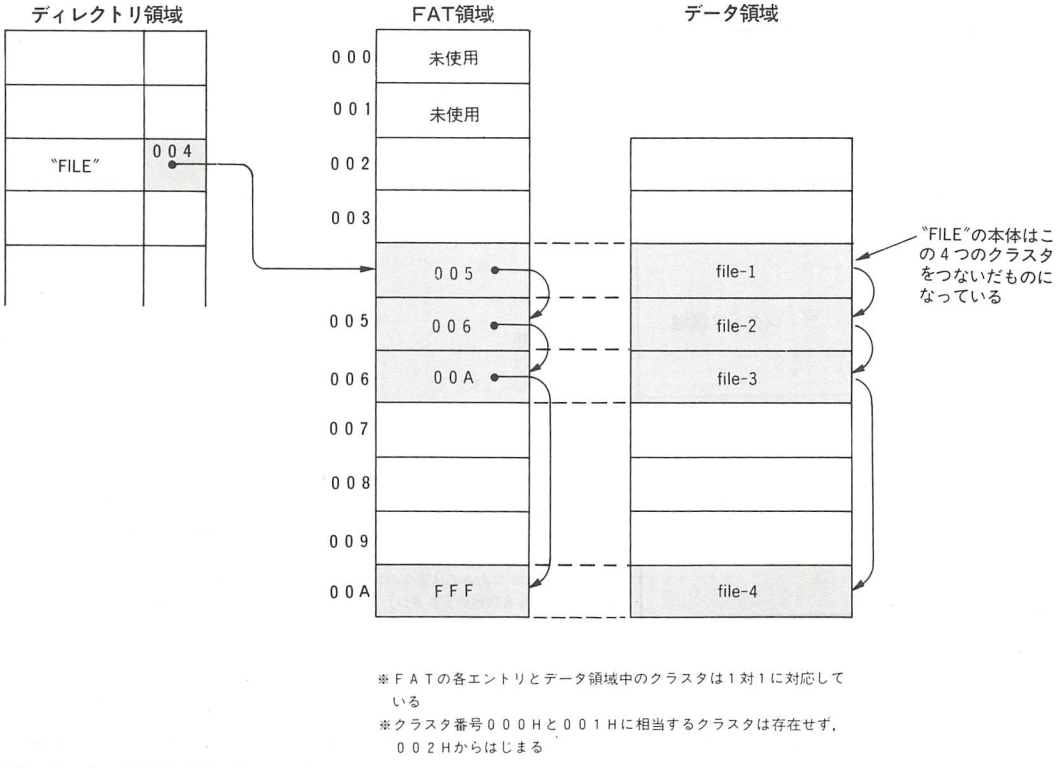


Fig. 5．8　FATとクラスタ

(FORMAT コマンドの一部は各メーカーで作られるため)．このような場合は，ディスクをフォーマットした後，どんな方法でもよいから，大きなファイルを１つ作り(残り容量が０になるくらいがベスト)，そのダミーファイルを RECOVER コマンドにかけます．すると，不良セクタの検出および除去を行ってくれるので，後はそのダミーファイルを消してしまえば不良セクタのないディスクが得られます(といっても，ディスクの初期不良のセクタが検出されるだけである．一般に１つでも不良セクタが発生するようなディスクでは，後々連鎖的に不良セクタが発生するということはよくある)．

5.3　ディスク上の領域

MS-DOS で扱うディスクは，Fig. 5.9 のように，４つの領域から成っています．

予約領域は通常ブートセクタと呼ばれ，ブート・プログラムを入れるために使われます．この領域はシステムの管理対象外であり，DEBUG でも使わない限りアクセスされることはありません．

FAT はすでに述べたように，ファイルやサブ・ディレクトリを構成しているクラスタのリンク状態や使用クラスタ，不良クラスタに関する情報を集めたテーブルのことであり，非常に重要な

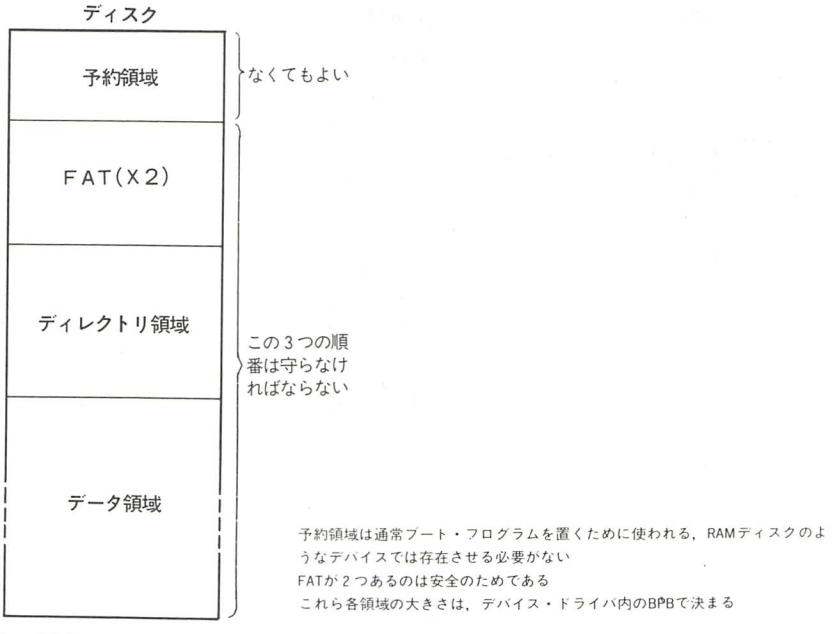


Fig. 5 . 9　ディスク上の領域

ものです．このうち(だいたい1〜4セクタで構成されている)一部のセクタでも欠けると，そのディスクはほぼ使用不可能となります．そのため，FATは2つのまったく同じコピーがディスク上に確保され，最初のものが使えなくなっても後の方を使うことができるようにしています．

ディレクトリ領域はすでに述べたルート・ディレクトリに相当するものです．

データ領域は，最初は未使用クラスタの集まりであり，必要に応じて先頭からファイルやサブ・ディレクトリのために順次割り当てられていきます．

以上4つの領域の大きさや，FATについてのさまざまな情報は，Table 5.2に示すようにディスクの種類ごとに決めてあるので，CP／Mのように，同じタイプのディスクでありながら互換性がないということはありません．

8インチと5インチ以外のディスク，たとえばハードディスクやマイクロフロッピーなどについては，まだ標準フォーマットは決まっていませんので，各メーカーにより異なる可能性があります．

5インチや8インチの倍密度のディスクで，セクタサイズが256バイトではなく，512バイトとか1024バイトになっているのは，セクタサイズを大きくしてディスク全体の容量を上げるためです．

8インチディスクは，インデックスホールの位置の違いにより片面か両面かの区別ができますが，5インチディスクではそうはいきません．5インチの場合，片面か両面か，8セクタ／トラックか9セクタ／トラックか，倍トラックか否かの組合せにより8通りのバリエーションがありま

Table 5．2　MS-DOS標準DISK FORMAT

	5インチ・ディスク								8インチ・ディスク		
トラック数	80	80	80	80	40	40	40	40	77	77	77
セクタ／トラック	9	9	8	8	9	9	8	8	26	26	8
サイド数	1	2	1	2	1	2	1	2	1	1	2
セクタ・サイズ	512	512	512	512	512	512	512	512	128	128	1024
ディスク総容量(Kバイト)	360	720	320	640	180	360	160	320	250	250	1230
ディレクトリ数	112	112	112	112	64	112	64	112	68	68	192
セクタ／クラスタ	2	2	2	2	1	2	1	2	4	4	1
予約セクタ	1	1	1	1	1	1	1	1	1	4	1
FAT数	2	2	2	2	2	2	2	2	2	2	2
セクタ／FAT	2	3	1	2	2	2	1	1	6	6	2
FAT ID	0F8H	0F9H	0FAH	0FBH	0FCH	0FDH	0FEH	0FFH	0FEH	0FDH	0FEH

すが，その区別は外見からは容易にはわかりません．そこで，MS-DOS ではFATの先頭の1バイト目に，Fig. 5.10 に示すようなFAT IDバイトと呼ばれる，ディスク識別用のデータを置いてあり，ディスクをアクセスする前に，このデータを読み出して，ディスクタイプを決定してからアクセスを行うようにしています．

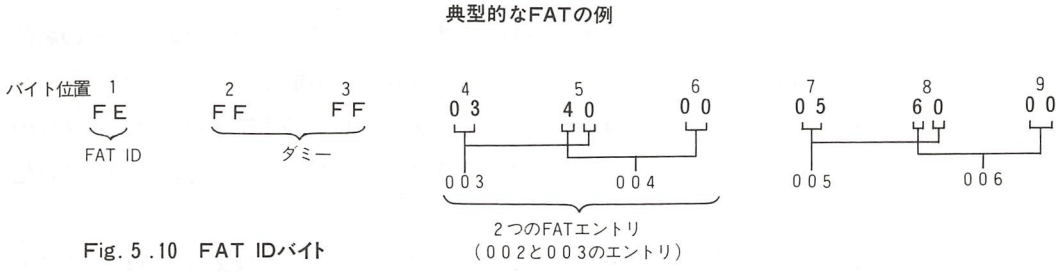


Fig. 5．10　FAT IDバイト

Table 5.2 からわかるように，FATの位置はどのディスクでも固定であるため，FATの読出しはどのタイプのディスクでも，同じ手続きで行えます．

なお，このFAT ID(と1ワードのFFH)がFATの先頭3バイトを占めているため，FATのエントリには000Hと001Hは存在せず，002Hからはじまります．そしてデータ領域の先頭のクラスタのクラスタ番号は002Hとなっています．

6章　マクロアセンブラ

6.1　マクロアセンブラの概要

マクロアセンブラは，MS-DOSの管理下で動作する8086／8088CPU用のセルフ・アセンブラで，以下のような特徴を持っています．

- リンカに入力可能なリロケータブルファイルを出力する
- マクロ機能を持つ
- セグメントの処理を効率的に行う
- 構造化されたデータを扱うことができる
- 8087演算プロセッサをサポートしている
- 各種の擬似命令を備えている

出力されるリロケータブルファイルは，マイクロソフト社の他の高級言語の出力するものと互換性を持っています．そのため，ひとつのプログラムを機械語や別々の言語で記述した複数のモジュールの集合体として作成することが可能です．

マクロ機能というのは，一連の命令をひとつの名前で表し，ソース・プログラム中でその名前が引用されるごとに，定義された命令に置き換えてアセンブルする機能です．一見サブルーチンと同じようですが，サブルーチンにするほど大きくない手続きや特定の手続きなどは，マクロの方が便利です．特にシステムコールは，プログラムの書きやすさ，見やすさなどの点でマクロ化すべきでしょう．

データの構造化というのは，複数のデータを1つのデータセットとして扱えるようにすることで，それにより，1つの名前で複数のデータを扱ったり，あるいはデータセットの一部のフィールドをアクセスすることが可能になります．とはいっても，ほかの高級言語のような厳密なデータセットではなく，擬似命令により表記を見やすく仕上げたり，あるいはデータの名前からオフセットを算出しているだけですが，結果的にはデータを構造化されているように扱うことができます．

このアセンブラは，8087演算プロセッサ(以下8087と略)の命令もアセンブルすることができます．また，8087を実装していなくても，8087エミュレート・ライブラリをリンカで連結することにより，8087の命令を使ってプログラムを記述することが可能です．この場合，アセンブラは8087用の命令を見い出すと，それをESC命令に変換するのではなく，外部の8087エミュレータ

を呼び出す命令に変換します．この機能はマクロアセンブラだけでなく，他の高級言語にも備えられています．8087 エミュレータは，8086CPU の命令で記述されたモジュールで，実行速度を除き 8087 と等価な処理を行います．

マクロアセンブラは，一般のアセンブラが持っている擬似命令のほかに，型を持ったデータを扱うことができるように，各種の擬似命令が用意されています．また，ほかにもセグメントの処理を容易にするための，セグメント関係の擬似命令も備えられています．

マクロアセンブラの機能があまりに豊富であるために，本章ではすべての機能の説明を行うことはできませんが，プログラムを動作させる上で最低限必要な点に関してはひととおり述べられています．

6.2　名前

アセンブラに限ったことではありませんが，プログラムを作成する際に，数値やアドレス，手続きを表すのに名前を使います．意味のある名前をこれらに付けることは，プログラムを理解しやすくするための強力な手段です．ここでは，名前の付け方，その種類について説明していきます．

6.2.1　名前

名前には，以下の文字を使用することができます．

- A～Z，a～z　（ただし，大文字と小文字は区別されない）
- 0～9
- ″_″，″$″，″?″，″@″

ただし，先頭は英字でなければなりません．長さは何文字でもかまいませんが，31 文字を超える分は認識されません．

6.2.2　名前の種類

アセンブラにおいて用いられる名前にはいくつかの用途がありますが，一般的なものは，プログラムやワークエリアに付けられるラベル，マクロ定義の名前，メモリには展開されない定数，変数を定義するシンボルといったところでしょう．マクロアセンブラでも，これらはもちろん備わっていますが，それ以外にもいくつか名前を付けられる，あるいは付ける必要のあるものがあります．ここでは，マクロアセンブラで扱う名前をひととおり説明しておきましょう．

ラベル

書式　〈名前〉：

プログラムの分岐先などを指定するための名前で，定義するときは〈名前〉の後ろに″：″を付けます．

プロシージャ

書式　〈名前〉 PROC 〈属性〉

サブルーチンなどの一連の手続きに対して付ける名前で，PROC 擬似命令と組み合せて使用します．別に使わなくても済むのですが，手続きやその属性を明確にするのに役立ちます．

ストラクチャ，レコード

書式　〈名前〉 STRUC／RECORD

ストラクチャ，レコードの定義に付けられる名前です．この名前で，メモリ中にその構造を持つ変数，定数を置くことができます．

メモリ中の変数，定数

書式　〈名前〉 〈定義〉

メモリ中に置く変数，定数に付ける名前です．

変数，定数

書式　〈名前〉 〈定義〉

メモリ中には置かれず，アセンブラ内のみで使われるもので，その名前自身に数値，文字などを定義します．

マクロ

書式　〈名前〉 MACRO 〈引数並び〉

マクロ定義に付けられる名前です．この〈名前〉をプログラム中で引用することにより，定義した命令に展開されます．

セグメント

書式　〈名前〉 SEGMENT〈属性〉

マクロアセンブラでは，プログラム，データなどを独立したセグメントとして扱いますが，これはそれに付ける名前です．それぞれのセグメントは，この〈名前〉で管理されます．

6.3 擬似命令

アセンブラは，ニーモニックで書かれたソース・プログラムを機械語に変換しますが(マクロアセンブラの場合は，リンカに入力するためのオブジェクトファイルに変換する)，すべての命令が変換されるわけではありません．変換されない命令には，擬似命令と条件アセンブルを指定された命令の2種類があります．

擬似命令というのは，8086／8088CPUが直接実行する命令ではなく，アセンブラ，あるいはリンカを制御するための命令です．また，条件アセンブルというのはソース・プログラム中の一部を条件判断によってアセンブルしたりしなかったりする機能で，擬似命令を組み合せることにより実現できます．

擬似命令には，メモリ制御，マクロ，条件アセンブル，リスティング制御などを指定するものがあります．

6.3.1 メモリ制御擬似命令

メモリ制御に関する擬似命令には，大きく分けて変数や定数をメモリ中に設定するものと，セグメントを管理するものの2種類があります．

変数や定数を設定する機能は従来のアセンブラにもありましたが，マクロアセンブラでは，構造化データの扱いなどのために，さらに機能が強化されています．また，命令のニーモニックで変数や定数の型を指定するようにはなっていないため，それらを擬似命令で定義するようになっています．アセンブリ言語で記述されたプログラムでは，メモリ中に定数を置いたり，作業用の領域を確保するために擬似命令を使いますが，マクロアセンブラのそれは，従来の8ビットCPUのものと比べ，意味あいがかなり異なっています．従来のものが，単に必要な領域を確保し，場合によってそこに値を収めていただけだったのに対し，マクロアセンブラでは，それに加えて，その変数，あるいは定数の型の指定まで行うのです．同様に，ラベルも従来のものは単に16ビットのアドレスを示す数値を持っていただけなのに対し，どのセグメントに属するか，どの型を持つかといった情報も持っています．これらの属性を以下に示します．

型

型といっても，整数，実数といったものではなく，それが何バイトを占めるかという属性です．

オフセット

アドレスに相当するものです．8086／8088CPUの場合，アドレスはセグメントベースとそこからのオフセットになりますから，そのオフセットを持つ必要があるわけです．ただし，マクロアセンブラの場合は，セグメントの組合せによりオフセットも変化しますから，実際の値はリンクするまでわかりません．

セグメント

オフセットとともに，アドレスを指定するために必要な情報です．ただし，実際のセグメントベースを持つのではなく，どのセグメントに属しているかという情報を持っています．実際のセグメントベースの値は，オフセットと同様に，リンクするまでわかりません．

次の例を見てください．Fig. 6.1 ①，②は，8080CPU と 8086／8088CPU において，2バイトの定数 const を定義するものです．これだけ見ると，どちらもたいした違いはないようですが，Fig. 6.1 ③，④までみるとだいぶ違うのがおわかりでしょう．同図③は，8080CPU で1バイトの定数2つに分解したもの，④は 8086／8088CPU で2つに分解したものです．③では前と同じ結果になっていたものが，④ではエラーになっています．このエラーを取り除くには，プログラムを⑤のように書き換え，型の指定を強制的に変更しなければなりません．

①

```
0000 2A0300             LHLD    CONST
                ;
0003 3412       CONST:  DW      1234H
                ;
0005                    END
```

②

```
0000                    code    segment
                        assume  cs:code,ds:var
0000  A1 0000 R                 mov     ax,const
0003                    code    ends
                        ;
0000                    var     segment
0000  1234              const   dw      1234h
0002                    var     ends
                        ;
                                end
```

③

```
0000 3A0300             LDA     CONST
                ;
0003 3412       CONST:  DW      1234H
                ;
0005                    END
```

④

```
0000                    code    segment
                        assume  cs:code,ds:var
0000  A0 0000 R                 mov     al,const   …ALレジスタ(8ビット)にワードデータをロードしようとしたのでエラーになった
E r r o r   ---         31:Operand types must match
0003                    code    ends
                        ;
0000                    var     segment
0000  1234              const   dw      1234h
0002                    var     ends
                        ;
                                end
```

⑤

```
0000                    code    segment
                        assume  cs:code,ds:var
0000  A0 0000 R                 mov     al,byte ptr const
0003                    code    ends
                        ;
0000                    var     segment
0000  1234              const   dw      1234h
0002                    var     ends
                        ;
                                end
```

変数の型をPTR演算子でバイトに変更．下位バイト（下位アドレス）がロードされる

Fig. 6.1　型指定8080vs8086CPU

○変数，定数

マクロアセンブラでは，変数，定数設定用の擬似命令として以下のものを用意しています．

- DB　(BYTE)　1バイト
- DW　(WORD)　2バイト
- DD　(DWORD)　4バイト
- DQ　(QWORD)　8バイト
- DT　(TBYTE)　10バイト

扱うことのできる数値には以下のものがあり，これらの数値を使えば，定数，あるいは変数の初期値の設定が行えます．8086／8088CPUは，基本的には1バイトか2バイトの整数しか扱えませんが，8087が10バイトの実数まで扱えるため，マクロアセンブラではそのための用意もされています．変数の場合，数値の代わりに〝?〟を使えば初期設定は行われません．

- 2進整数　01000011B
- 8進整数　177Q
- 10進整数　16384
- 16進整数　0FE01H
- 10進実数　1.23E5
- 16進実数　FEDCBA9876543210R
- ASCIIキャラクタ　'ABCDEFGHIJK'

これらを使った変数，定数の設定をFig. 6.2に示します．この中に出てくる〝DUP( )〟というのは，設定の繰返し指定で，これにより，同じ型の変数をまとめて定義することができます．つまり，配列変数が定義できるわけです．

```
0000                                    var     segment
                                                assume  ds:var
                                        ;
0000  B6                                bin     db      10110110b ……2進数でバイトデータを設定
0001  1FFF                              oct     dw      17777q ……8進数でワードデータを設定
0003  40 E2 01 00                       deci    dd      123456 ……10進数で2ワードデータを設定
0007  CD AB 89 67 45 23                 hex     dq      123456789abcdh …16進数で4ワードデータを設定
      01 00
000F  C0 04 6E DD CD 53                 real    dt      3.141592e01 ……10進数で5ワードデータを設定
      7B 85 00 20
0019  4D 53 2D 44 4F 53                 char    db      'MS-DOS' ……文字でバイト列を設定
001F     10 [                           array   dw      16 dup(?) ……16ワードの領域を確保
              ????
                   ]

                                        ;
003F                                    var     ends
                                                end
```

Fig. 6．2　変数，定数の定義

○ストラクチャ

マクロアセンブラでは，前述のnバイト変数，定数以外に，構造化されたデータを扱うことが可能になっています．これは，複数のデータから構成される1組のデータセットに名前を付け，一括してアクセスできるようにしたもので，このデータセットのことをストラクチャと呼びます．たとえば，部品番号(number)，名前(p_name)，価格(price)の3つのフィールドから構成されるストラクチャ〝part〟を定義し，それを使っていくつかのストラクチャを設定してみましょう(Fig. 6.3)．

```
0000                                code    segment
                                    assume  cs:code,ds:var
0000  A1 0000 R                     mov     ax,engine.number ……engineというストラクチャのnumberという要素をAXレジスタにロード
0003                                code    ends
                                    ;
0000                                var     segment
                                    ;
                                    part    struc ……partというストラクチャの定義
0000  ????                          number  dw      ? ……number ワード
0002  20 20 20 20 20 20            p_name  db      '                    '
      20 20 20 20 20 20                             ……p_name 20文字のバイトデータ
      20 20 20 20 20 20
      20 20
0016  ????????                      price   dd      ? ……priceという2ワードデータ
001A                                part    ends
                                    ;
0000  0001                          engine  part    <1,'gas turbine engine',10000>
0002  67 61 73 20 74 75                              └── engineというpartで定義されたストラクチャを設定
      72 62 69 6E 65 20
      65 6E 67 69 6E 65
      20 20
0016  10 27 00 00

001A  0002                          gear    part    <2,'left main gear',3500>
001C  6C 65 66 74 20 6D
      61 69 6E 20 67 65
      61 72 20 20 20 20
      20 20
0030  AC 0D 00 00

0034  0003                          tyre    part    <3,'rubber tyre',150>
0036  72 75 62 62 65 72
      20 74 79 72 65 20
      20 20 20 20 20 20
      20 20
004A  96 00 00 00

                                    ;
004E                                var     ends
                                            end
```

Fig. 6.3　パーツのリストその①(ストラクチャの構造)

Fig. 6.3では，〝part〟がストラクチャの構造に対して付けられた名前で，これ自身はメモリ中には展開されません．実際にデータとしてメモリ中に展開されるのは，それ以降の〝engine〟，〝gear〟，〝tyre〟といった名前を付けられた部分です．これらは，それぞれが〝part〟と名付けられた構造を持つストラクチャとなります．これらを参照する場合，ストラクチャの名前によりそ

の先頭のアドレスが，フィールド名により各フィールドのオフセットが得られます．もし，これらをストラクチャを使わずに表すと，Fig. 6.4 のように，プログラムは長くてわかりにくいものになってしまいます．

```
0000                                ;
                                    code    segment
                                            assume  cs:code,ds:var
0000  A1 0000 R                             mov     ax,engine+number
0003                                code    ends
                                    ;
0000                                var     segment
                                    ;
0000  0001                          engine  dw      1
0002  67 61 73 20 74 75            p_name$ db      'gas turbine engine  '
      72 62 69 6E 65 20
      65 6E 67 69 6E 65
      20 20
0016  ????????                      price$  dd      ?
                                    ;
001A  0002                          gear    dw      2
001C  6C 65 66 74 20 6D            db      'left main gear      '
      61 69 6E 20 67 65
      61 72 20 20 20 20
      20 20
0030  AC 0D 00 00                           dd      3500
                                    ;
0034  0003                          tyre    dw      3
0036  72 75 62 62 65 72            db      'rubber tyre         '
      20 74 79 72 65 20
      20 20 20 20 20 20
      20 20
004A  96 00 00 00                           dd      150
                                    ;
004E                                var     ends
                                    ;
= 0000                              number  equ     0
= 0002                              p_name  equ     p_name$-engine
= 0016                              price   equ     price$-engine
                                    ;
                                            end
```

各要素のオフセットの設定

Fig. 6．4　パーツのリストその②

○レコード

レコードは，ストラクチャに似ています．ただ，ストラクチャがバイト以上の単位でフィールドを設定するのに対し，レコードは，ビット単位でフィールド設定します．

たとえば，3 ビットのデータ 5 つをまとめて管理したいとき，1 バイトにデータを 1 つしか収めないのでは，メモリ効率が悪くなります．このデータは全部で 2 バイトに収まりますから，パックしておく方がメモリ効率がよいのは明らかですが，各フィールドをアクセスするのはめんどうなものになります．マクロアセンブラには，このような処理を楽に行えるようにするための擬似命令，演算子が用意されています．今の例で，5 つのフィールドをそれぞれ〝fl1〟～〝fl5〟とし，それを収める 2 バイトを〝packed〟としたときの設定，各フィールドの読み書きのマクロ定義〝getf〟，〝putf〟を Fig. 6.5 に示します．

```
                                putf    macro   rec_name,x,putdata
                                        mov     cl,fl&x
                                        shl     putdata,cl
                                        mov     ax,mask fl&x
                                        not     ax
                                        and     ax,rec_name
                                        or      ax,putdata
                                        mov     rec_name,ax
                                        endm
                                ;
                                getf    macro   rec_name,x
                                        mov     cl,fl&x
                                        mov     ax,rec_name
                                        and     ax,mask fl&x
                                        shr     ax,cl
                                        endm
                                ;
0000                            code    segment
                                        assume  cs:code,ds:var
                                        putf    packed,3,init
0000  B1 06 90 90             +         mov     cl,fl3
0004  D3 26 0002 R            +         shl     init,cl
0008  B8 01C0                 +         mov     ax,mask fl3
000B  F7 D0                   +         not     ax
000D  23 06 0000 R            +         and     ax,packed
0011  0B 06 0002 R            +         or      ax,init
0015  A3 0000 R               +         mov     packed,ax
                                        getf    packed,3
0018  B1 06 90 90             +         mov     cl,fl3
001C  A1 0000 R               +         mov     ax,packed
001F  25 01C0                 +         and     ax,mask fl3
0022  D3 E8                   +         shr     ax,cl
0024                            code    ends
                                ;
                                pack    record  fl1:3,fl2:3,fl3:3,fl4:3,fl5:3
                                ;
0000                            var     segment
0000  00 00                     packed  pack    <0,0,0,0,0>
0002  0005                      init    dw      5
0004                            var     ends
                                ;
                                        end
```

5つの3ビットデータから構成されるレコードpackを定義

packedという名前のレコードを設定

Fig. 6.5　レコードの定義

各フィールドの抽出や置換えは，1命令で処理することはできませんが，レコードの定義と，各種演算子の働きで，すっきりと書くことができます．具体的には，MASK 演算子による着目フィールドの抽出，各フィールド名から得られるシフトカウント，論理演算を組み合せることにより，必要とするフィールドの抽出，セットが行えるわけです．

このようにレコードを使うことにより，ビットフィールドの扱いはマクロアセンブラではかなり容易になっています．しかし，8086／8088CPU が基本的に 16 ビットデータまでしか扱えないため，レコード長は，最大2バイトまでしか設定できません．

6.3.2　ラベル

プログラムの分岐に関するラベルは，変数のような型は持ちませんが，その代わりにどのセグメントにあるラベルなのかという情報があります．それにより，そのラベルへ分岐するときに，同一セグメント内であれば NEAR(IP のみの変更)，異なるセグメントであれば FAR(IP と CS

を両方とも変更)というようになります．ラベルの持つ属性を以下に示します．

オフセット

そのラベルのあるセグメント内でのオフセットです．ただし，実際の値はリンクするまでわかりません．

セグメント

そのラベルのあるセグメントの名前です．実際のセグメントベースはリンカによって決定されます．

セグメントを管理する擬似命令は，8086／8088CPUが1Mバイトのメモリ空間を管理するために，セグメントという概念を持ち込んだために必要になったものです．この管理は，セグメント・レジスタを操作するようなプログラムになるとかなり手間のかかるものになりますが，その負担を少しでも軽減すべく，アセンブラとリンカの機能が豊富になっています．セグメントに関する擬似命令は，アセンブラやリンカに必要な情報を与えるために使われます．これに関しては，次のセグメントの扱いのところで説明します．

6.3.3　マクロ機能用擬似命令

○マクロ

マクロ機能は，さきにも説明したように，一連の命令を1つの名前で参照できるように定義するものです．マクロ用の擬似命令は，主にその定義に関するものです．

マクロが実際にどのように使われるかを見てみましょう．Fig. 6.6がマクロの定義とその引用です．

```
                                add_stc macro
                                        pop     ax
                                        pop     dx
                                        add     ax,dx
                                        push    ax
                                        pushf
                                        endm
                                ;
0000                            code    segment
                                        assume  cs:code
                                        add_stc
0000  58                        +       pop     ax
0001  5A                        +       pop     dx
0002  03 C2                     +       add     ax,dx   } マクロ展開により生成されたコード
0004  50                        +       push    ax
0005  9C                        +       pushf
0006                                    ends
                                ;
                                        end

                マクロ展開の印
```

Fig. 6．6　マクロの定義と引用

Fig. 6.6のマクロ定義は，スタックに収められている2つの16ビットデータをポップし，その加算した結果の16ビットとフラグをスタックに収めるというもので，〝add_stc〟という名前で定義してあります．この手続きを一度マクロ定義しておくことで，後はプログラム中で〝add_stc〟という名前で引用できることがおわかりでしょう．また，Fig. 6.7のように，マクロを展開するときにパラメータを渡すことも可能です．これは，文字列出力を行うシステムコールのマクロ定義で，〝string〟の名前が付いているアドレスから〝$〟の手前までを画面に出力します．

```
                                display macro   string ……パラメータ
                                        mov     ah,09h
                                        mov     dx,offset string
                                        int     21h
                                        endm
                                ;
0000                            code    segment
                                        assume  cs:code,ds:var
                                ;
                                        display message
0000  B4 09                     +       mov     ah,09h
0002  BA 0000 R                 +       mov     dx,offset message
0005  CD 21                     +       int     21h
                                ;
0007                            code    ends
                                ;
0000                            var     segment
0000  54 68 69 73 20 69         message db      'This is a pen.'
      73 20 61 20 70 65
      6E 2E
000E  0D 0A                             db      0dh,0ah
0010  24                                db      '$'
0011                            var     ends
                                ;
                                        end
```

Fig. 6.7　システムコールのマクロ定義

この2つの例が標準的なマクロの利用法ですが，他の機能と組み合せると，マクロはさらに強力なものとなります．たとえば，マクロ定義の中で後述の条件アセンブルを行えば，引用されたときの条件に応じて異なった命令に展開することが可能です．

○繰返し

マクロとは少し異なりますが，繰返し用の擬似命令というものもあります．これは，指定した回数，あるいは不定回数ある処理を繰り返すための機能で，ソース・プログラムが短く見やすいものになります．

Fig. 6.8は，〝bit0〟～〝bitF〟までの16個の16ビット定数を定義するものです．繰返し用の擬似命令を使用することにより，16個並べて書くより簡単に書けるようになります．

```
                                ;
0000                            var     segment
                                        assume  ds:var
= 0001                          bit     =       0000000000000001b
                                        irpc    x,0123456789ABCDEF  ｝Xを0～Fで置き換え
                                bit&x   dw      bit                 ｝bitとつなぐ
                                bit     =       2*bit
                                        endm
0000  0001                    + bit0    dw      bit
0002  0002                    + bit1    dw      bit
0004  0004                    + bit2    dw      bit
0006  0008                    + bit3    dw      bit
0008  0010                    + bit4    dw      bit
000A  0020                    + bit5    dw      bit
000C  0040                    + bit6    dw      bit
000E  0080                    + bit7    dw      bit
0010  0100                    + bit8    dw      bit
0012  0200                    + bit9    dw      bit
0014  0400                    + bitA    dw      bit
0016  0800                    + bitB    dw      bit
0018  1000                    + bitC    dw      bit
001A  2000                    + bitD    dw      bit
001C  4000                    + bitE    dw      bit
001E  8000                    + bitF    dw      bit
0020                            var     ends
                                ;
                                        end
```

Fig. 6．8　繰り返し用擬似命令

この中に〝&〟という記号がありますが，これは，マクロ定義や繰返しの中だけで使用可能な演算子で，文字列の連結を行います．Fig. 6.8 では，〝bit〟と仮引数の〝x〟をこれで連結しています．〝x〟は展開のときに 0～F に置換されますから，〝bit〟と〝x〟が文字列として連結され，異なる 16 個の名前を生成するわけです．

この例は，DW 擬似命令の繰返しでしたが，もちろん，命令の繰返しにも使うことが可能です．この場合も，繰り返すごとに少しずつ異なる命令を生成することができます．

6.3.4　条件アセンブル擬似命令

条件アセンブルというのは，アセンブル中に条件判断を行い，その結果によりソースリストの一部分をアセンブルしたりしなかったりする機能です．当然，条件判断はアセンブル中に得られる情報に限定されますから，プログラム中のレジスタの値やフラグによる判断などとは異なります．プログラム中の条件判断によるスキップには，実行されないコードが存在していますが，条件アセンブルはスキップされる部分のオブジェクトは出力されません．その結果，指定した条件によって，異なるオブジェクトを出力することになります．

たとえば，あるプログラムでメッセージの表示を和文で行うか，英文で行うかの判断を考えてみましょう．このような場合，普通はシステムのワークエリアを調べればそのシステムが漢字を扱えるかどうかがわかりますから，プログラム内の条件判断でどちらにするかを決定することができます．あるいは，実行時に与えられるパラメータにより決定することも可能です．しかし，好み，あるいはハードウェアの制約で，普通はどちらかに固定されるでしょう．このような場合，

プログラムはどちらにでも対応できる必要はまったくなく，使われない部分はメモリのむだ以外の何物でもありません．かといって，使われていない機能をソース・プログラムから削除して2つのプログラムのソースを取っておいたり，あるいは，1つのプログラムを必要に応じて編集し直すというのも問題です．このようなときに便利なのが条件アセンブルの機能で，これを使うとソース・プログラム内には両方の機能を用意しておき，条件に応じてアセンブルを行い，必要なオブジェクトのみを得ることができます．

このような条件アセンブルは，同じプログラムをフラグを変更するだけで仕様の異なるシステムに載せるときなどに行われます(Fig. 6.9)．

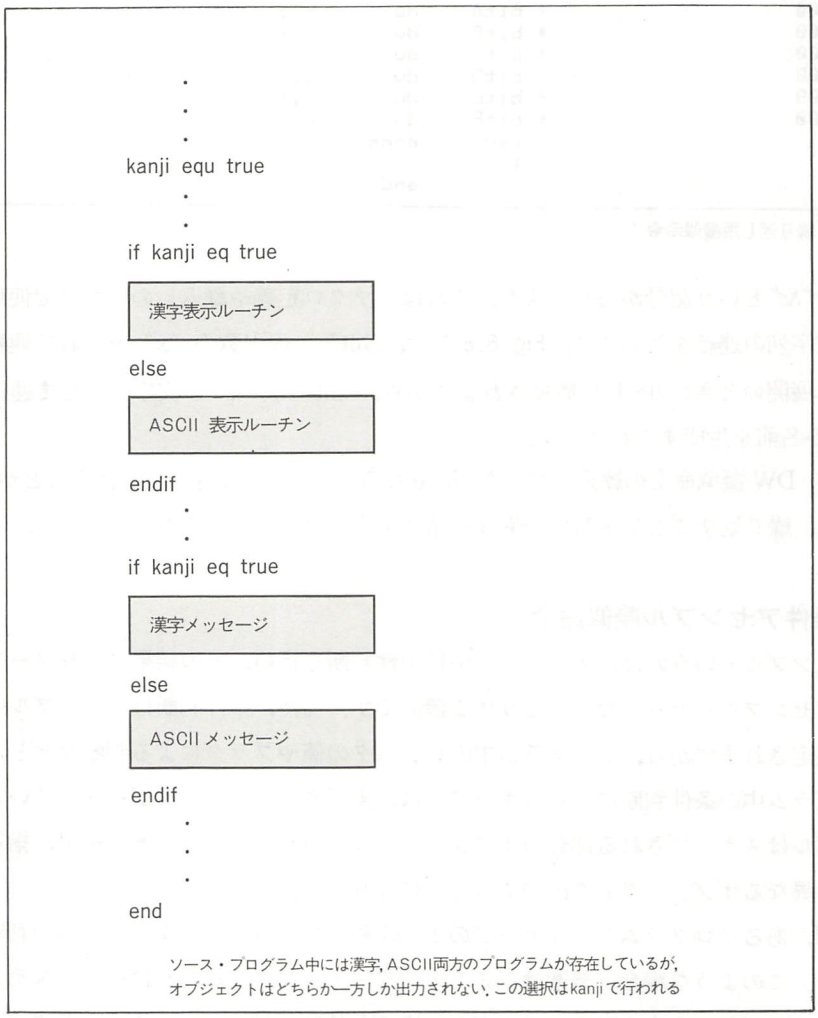


Fig. 6．9　条件アセンブル

条件アセンブルは，マクロ機能と組み合せるとさらに有効に利用できます．マクロの定義の中で条件アセンブルを指定しておき，そのマクロを引用するときに条件を設定すれば，1つのマクロ定義で，異なった命令群を生成することができます．通常，異なる手続きを必要とする場合は別々のマクロとして定義すれば問題ないのですが，大きなマクロ定義のうちごく一部のみが異なる場合や引用時の条件により展開のされ方が異なる場合などは，条件アセンブルと組み合せた方が便利です．Fig. 6.10 は，マクロのパラメータにより異なったオブジェクトを出力する条件アセンブルの例です．

```
0000                            var     segment
                                        assume  cs:code,ds:var
0000  E7                        b_data  db      -25
0001  4C2C                      w_data  dw      19500
0003                            var     ends
                                ;
                                add_ext macro   x
                                        if      type x eq 1     ……xがバイト型なら以下をアセンブルする
                                ;       x is byte
                                        mov     al,x
                                        cbw     ……ALレジスタを符号拡張し，AXレジスタに収める
                                        else    }
                                ;       x is word }xがバイト型でなければ以下をアセンブルする
                                        mov     ax,x
                                        endif
                                        add     ax,dx
                                        endm
                                ;
0000                            code    segment
                                        add_ext b_data
0000  A0 0000 R                 +       mov     al,b_data
0003  98                        +       cbw
0004  03 C2                     +       add     ax,dx
                                        add_ext w_data
0006  A1 0001 R                 +       mov     ax,w_data
0009  03 C2                     +       add     ax,dx
000B                            code    ends
                                ;
                                        end
```

Fig. 6.10　条件アセンブルとマクロ

この例は，パラメータとなる変数 x を DX レジスタに加算して，その結果を AX レジスタに収めるというものです．このとき，x がワードであればそのまま，バイトのときは 16 ビットに符号拡張してから行います．この符号拡張の処理はプログラム中のスキップではなく，変数の型による条件アセンブルですから，展開時には不要なオブジェクトは出力されません．

6.3.5　リスティング擬似命令

リスティング擬似命令は，主にアセンブル出力の書式などの指定するのに使用され，オブジェクト出力にはほとんど関係しません．ですから，これらはまったく使用しなくても結果には影響しませんが，プログラムの開発環境を整えるには便利です．これらをうまく利用することにより，アセンブル時の繁雑なオプションの設定を省略したり，リスト中にきれいなコメントやタイトルを入れることができます．

6.3.6　その他の擬似命令

マクロアセンブラには，以上で説明してきたもの以外にもいくつかの擬似命令があります．これらはマクロアセンブラに特有なものではなく，ごく一般的なものですので，簡単に説明しておきます．

ORG

ロケーション・カウンタ(現在の着目アドレスを示すポインタ)の値を変更します．

EQU

シンボルに値を定義します．これはDB, DWなどと異なり，メモリ中には置かれません．また，再定義はできませんから，その必要がある場合は，次の〝=〟を使います．

=

EQUと同じように，シンボルに値を定義しますが，再定義することができます．引用した場合は，その直前で定義された値が返されます．

PUBLIC

名前が外部モジュール(リンカでリンクされる別のモジュール)から参照されるものであることを宣言します．

EXTRN

名前が外部を参照しているものであることを宣言します．このとき，その名前は外部モジュールでPUBLICの宣言を受けている必要があります．

INCLUDE

外部のソースファイルを読み込み，それをモジュールの一部としてアセンブルします．システムコールのマクロ定義などの使用頻度の高いモジュールを毎回組み込むより，ソースファイルを小さくできます．

.RADIX

数値の基数を変更します．通常，基数を省略した場合は10進数とみなされますが，これを使用すると，省略時の基数を2から16までの間に設定できます．

END

ソース・プログラムの終了をアセンブラに伝えます．〝END〟の後ろにラベルを書くと，そこがスタート・アドレスになります．

6.4　セグメントの扱い

8086／8088CPU は，アドレス空間を 64K バイト以上に拡大するためにセグメントという概念を導入したわけですが，機械語レベルでプログラムを作成する場合，次に示すような点を考慮する必要があるため，かなり扱いが複雑なものとなります．

- １つの物理アドレスに対して複数の論理アドレスが存在する
- 用途により異なる４つのセグメントがある
- １つのセグメントの大きさは，セグメントベースを変更しない限り 64K バイトまでしかない

これらは，上限 64K バイトという点を除き長所ともなり得る点ですが，やはりリニアアドレスの CPU にはないわずらわしさがあります．ここでは，プログラムを作成するときにセグメントをどのように扱えばよいか,また,アセンブラがセグメントをどのように扱うかについて説明します．

6.4.1　8086／8088 CPUのセグメント

8086／8088CPU には，コード，データ，エキストラ，スタックの４つのセグメントがあり，それらは命令によって適当なものが選択されます．アドレスの指定はセグメント・レジスタから得られるセグメントベースと，そこからのオフセットという２つの値を加算することによって得られます．また命令の多くは，必要があれば１バイトのプリフィックスにより，セグメントをデフォルトのものから変更することが可能です．

8086／8088CPU のプログラムの標準的なメモリマップは Fig. 6.11 のようになります．プログラムはコード・セグメント内，スタックはスタック・セグメント内，データはデータ／エキストラ・セグメント内というものです．これらのセグメントは，状況によって重なる部分があったりなかったりします．また，物理アドレス内におけるセグメントの位置は，プログラム内で変更することが可能です．

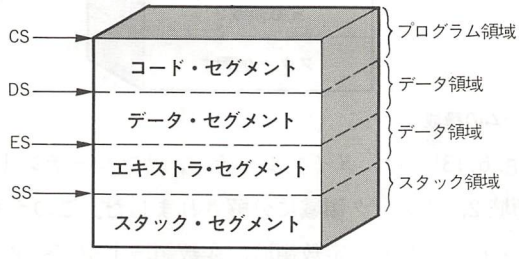


Fig. 6 .11　8086／8088CPUの平均的メモリマップ

このように，8086／8088CPUは，独立した4つのメモリ空間を持っているかのように動作します．しかし，アセンブラにかけるプログラムは1次元的なものですから，処理や定義がどのセグメントに対してのものなのかをプログラム中で記述しておかなければなりません．これらは，擬似命令により指定されます．

6.4.2　アセンブラにおけるセグメントの扱い

もし，プログラム中において各セグメントが64Kバイトを超えず，その結果セグメントベースを変更することもなく，また，セグメントのオーバーライド指定(デフォルトの指定でなく，強制的に変更すること)もなければ，プログラムの作成は簡単なものになります．単に64Kバイトの空間を4個持つだけなので，各セグメント内においては，従来のアセンブラと同様の処理を行うだけで済みます．しかし，大きなプログラムとなると，そうもいっていられません．ここでは，マクロアセンブラがどのようにセグメントを扱っているのかを説明しましょう．

○セグメントの分割とグループ

まず，アセンブリ言語で記述されたプログラムがどのような構造を持っているかを見てみましょう．一般にアセンブリ・プログラムは，Fig. 6.12のように，手続きを記述した部分と，その手続きにより処理されるデータ，ワークエリアなどから構成されます．リニアアドレスのCPUの場合，これらは同じ空間内に置かれますが，8086／8088CPUでは，それぞれが別のセグメント内に置かれます．とりあえず，これらがどのセグメントに関係しているかということを意識しながら分解してみましょう．

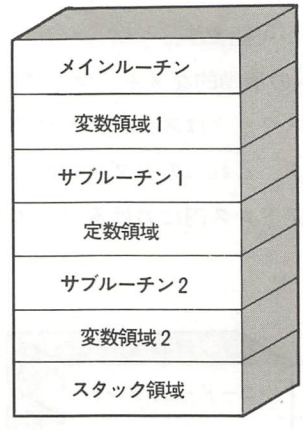


Fig. 6 .12　アセンブリ・プログラムの構成

このプログラムの例(Fig. 6.13)では，メインルーチン，サブルーチン1，サブルーチン2，定数領域，変数領域1，変数領域2，スタック領域に分解されました．このうち，メインルーチン，サブルーチン1，2はコード・セグメント，定数領域，変数領域1がデータ・セグメント，変数領域2がエキストラ・セグメントに置かれるとしましょう．そこで，同じセグメント内に連続して置か

```
                                ;       group
                                ;
                                subr    group   sub1,sub2
                                work1   group   const,var1
                                ;
                                ;       main routine
                                ;
0000                            main    segment para
                                        assume cs:main
0000  90                                nop                     ┐
                                ;        :                      │
                                ; .      :                      ├ メインルーチン
0001                            main    ends                    ┘
                                ;
                                ;       subroutine No.1
                                ;
0000                            sub1    segment para public 'subroutine'
                                        assume  cs:subr
0000                            subr1   proc    far             ┐
0000  90                                nop                     │
                                ;        :                      │
                                ;        :                      ├ サブルーチン1
0001  CB                                ret                     │
0002                            subr1   endp                    │
0002                            sub1    ends                    ┘
                                ;
                                ;       constant                ┐
                                ;                               │
0000                            const   segment para public 'work1'
                                        assume  ds:work1
0000  01                        cnst1   db      1               ├ 定数領域
                                ;        :                      │
                                ;        :                      │
0001  0F                        cnstn   db      15              │
0002                            const   ends                    ┘
                                ;
                                ;       subroutine No.2         ┐
                                ;                               │
0000                            sub2    segment para public 'subroutine'
                                        assume  cs:subr
0000                            subr2   proc    far             ├ サブルーチン2
0000  90                                nop                     │
                                ;        :                      │
                                ;        :                      │
0001  CB                                ret                     │
0002                            subr2   endp                    │
0002                            sub2    ends                    ┘
                                ;
                                ;       variable
                                ;
0000                            var1    segment para public 'work1'
0000  ??                        var11   db      ?
                                ;        :
                                ;        :
0001                            var1    ends
                                ;
                                ;       variable No.2
                                ;
0000                            var2    segment
                                        assume  es:var2
0000  ????                      var21   dw      ?
                                ;        :
                                ;        :
0002                            var2    ends
                                ;
                                ;       stack
                                ;
0000                            stc     segment para stack
                                        assume  ss:stc
0000     40 [                           dw      64 dup(?)
              ????
                  ]

0080                            stc     ends
                                ;
                                        end
```

Fig. 6.13　マクロアセンブラにおけるセグメントの例

れる一連のプログラムなりデータなりを1つのセグメントとして扱うことにします(ここでいうセグメントとは，8086／8088CPUの物理的なセグメントとは別のもの)．この例では，メインルーチン，サブルーチン1，2はそれぞれ別のセグメントとなるわけです．マクロアセンブラでは，プログラムはこのようなセグメントの集合体として扱われます．ただし，このままでは扱いにくいので，これらのセグメントに名前を付け，それぞれは名前で参照できるようにします．SEGMENT擬似命令とENDS擬似命令で囲まれた部分が，そのセグメントです．また，そのセグメントが8086／8088CPUのどのセグメントとして扱われるかも指定する必要がありますから，それをASSUME擬似命令で行います．

このようにしてプログラムを小さなセグメントの集合体としたわけですが，このままでは，分割単位が小さすぎます．これらのうちのいくつかは，1つにまとめて同じセグメントへ置くことができますから，それをグループ(GROUP)としてまとめます．ここでは，サブルーチン1，2をSUBR，変数領域1と定数領域をWORK1でまとめることにします．変数領域2とスタック領域はまとめず，それぞれエキストラ・セグメントとスタック・セグメントに置きます．このように設定した場合のセグメントごとのメモリマップはFig. 6.14のようになります．

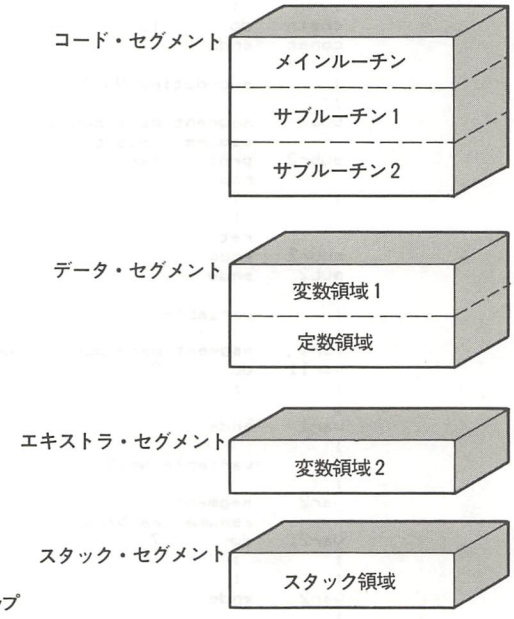


Fig. 6.14　セグメントごとのメモリマップ

○セグメントの配置

前に説明したように，8086／8088CPUの各セグメントは，重ねることも独立して置くことも可能です．これらの関係も明確に設定し，きめの細かい処理を行うためにSEGMENT擬似命令があります．SEGMENT擬似命令によって指定できるセグメントの配置には，プライベート(デフォルト)，PUBLIC，STACK，COMMON，MEMORYがあり，それらを指定したときのセグメントの関係は以下のとおりです．

プライベート

プライベートと指定されたセグメントは，完全に独立したセグメントとして扱われます．つまり，ほかのセグメントとはいっさい重なる部分も持たず，また同じ名前を持つ複数のセグメントがあっても，それぞれの間にまったく関係はなく別々のセグメントベースを持ちます．SEGMENT 擬似命令で何も指定しないとプライベートになります(Fig. 6.15)．

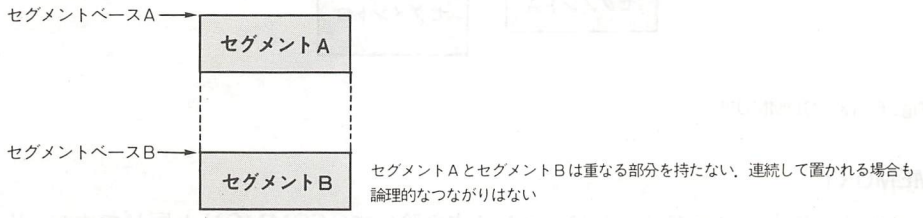


Fig. 6．15　プライベートセグメント

PUBLIC

同じ名前，あるいは後述するクラス名を持つセグメントを，まとめて1つのセグメントとします．まとめられたセグメントは，共通のセグメントベースを持ち，セグメント内のアドレスの指定は，そのセグメントベースからのオフセットで行われます(Fig. 6.16)．

PTR は変数の型以外に，ラベルの NEAR，FAR の指定も行えます．



Fig. 6．16　PUBLIC

STACK

スタック・セグメント用であることを除いて，基本的には PUBLIC と同じです．この指定をすると，SP(スタック・ポインタ)にこのセグメントの合計した長さが初期設定されますから，スタック領域は，このセグメントの最後から取られることになります(Fig. 6.17)．



Fig. 6．17　STACK

COMMON

同じ名前，あるいはクラス名を持つセグメントを同じアドレスからオーバーラップして設定します．当然セグメントベースは共通のものになります．これを指定すると，ひとつの物理アドレスに対して，複数の変数，定数を割り当てることが可能になります(Fig. 6.18)．

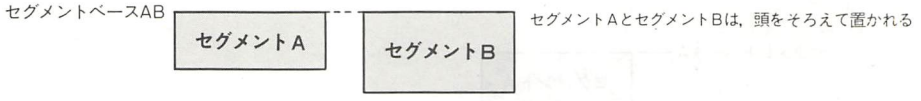


Fig. 6 .18　COMMON

MEMORY

メモリの最上位にセグメントが置かれる点を除いて，COMMON と同じですが，リンカでサポートされていませんから，MS-DOS 上では意味がありません．

AT 〈式〉

セグメントを〈式〉で指定されるパラグラフ・アドレスから配置します．パラグラフ・アドレスというのは，xxxx0H と指定されるアドレスで，セグメント・レジスタに xxxxH を設定したときにオフセット 0000H となるアドレスです．

MS-DOS の管理下にある場合，この機能はプログラムをロードするアドレスを指定する用途には使えません．主に，変数やラベルにシステム内の特定のアドレスを割り当てるような用途に使用します．

○セグメント名とクラス名

セグメントの組合せは，セグメント名，クラス名によって行われます．セグメント名は，組合せの指定以外にもいくつか用途があるので省略できませんが，クラス名は省略可能です．同じクラス名を持つセグメントは，リンカによってメモリ中に連続してロードされ，その順番はソース・プログラムに現れた順番となります．つまり，クラス名を使うことにより，オブジェクトの配置を意識的に制御できるわけです．このことから，セグメント名はセグメントの組合せ以外の指定に，クラス名はセグメントの組合せの指定に使うとプログラムがわかりやすくなります．

6.5　演算子

マクロアセンブラは前述のように，基本的に変数や定数が型を持っているアセンブラで，命令中ではほとんど型の指定などは行いません．そのため，アセンブラによるプログラムの記述の大きなメリットの１つである，プログラムとデータの区別，データの型の違いによる区別などの制

約をなくした処理が，そのままでは不可能です．そこで，アセンブラらしい処理を実現するために，命令中で指定することによりデフォルトの指定(型の定義などの擬似命令で行われる指定)を無効にしたり，あるいはその指定を命令中で明確にすることができるようになっています．

マクロアセンブラに用意されている演算子のうちのいくつかは，このような処理に用いられます．もちろん，加算，減算などのいわゆる演算処理のための演算子も用意されています．

6.5.1　アトリビュート演算子

アトリビュート演算子は，型を持った変数やラベルの属性の抽出を行います．これにより，型指定の変更などが行えます(Table 6.1)．

Table 6．1　アトリビュート演算子 機能一覧

演算子	機　能
PTR	オペランドの型を指定，あるいは変更する
:	アドレスのセグメント指定を変更する
SHORT	セグメント内の相対分岐が+129～−126の範囲であることを宣言する
THIS	指定された属性を持つオペランドを作成する
HIGH	16ビットの上位8ビットを返す
LOW	16ビットの下位8ビットを返す
SEG	変数，ラベルのセグメントのベース・アドレスを返す
OFFSET	変数，ラベルのオフセットを返す
TYPE	変数のバイト数，ラベルのNEAR，FARを返す
.TYPE	変数のモード，外部参照の有無を調べる
LENGTH	その型の変数がいくつ定義されているかを返す
SIZE	その変数が合計何バイト割り当てられているかを返す
フィールド名	レコード内のそのフィールドの最下位ビットの位置を返す
WIDTH	フィールドの幅をビット数で返す
MASK	フィールド位置に1が立っているワードを返す

ここで使用頻度の高いアトリビュート演算子について簡単に説明しましょう．

PTR

書式　〈型〉 PTR 〈名前〉

〈名前〉の型を指定します．この演算子は，主に異なる型で定義されている変数，ラベルの指定に使われます．たとえば，バイトで定義されている変数をワードでアクセスするような場合，Fig. 6.19のようになります．

```
0000                            code    segment
                                        assume  cs:code,ds:var
0000  A1 0000 R                         mov     ax,word ptr string
0003                            code    ends            ……string(本当はバイト型)からワードでロードする
                                ;                       AXレジスタには6261Hが収められる
0000                            var     segment
0000  61 62 63 64 65 66         string  db      'abcdef'
0006                            var     ends
                                ;
                                        end
```

Fig. 6 .19　PTR

書式　：

〈セグメント指定〉：〈アドレス〉

セグメント・レジスタの指定をデフォルトのものから変更します．〈セグメント指定〉は，セグメント・レジスタ名，セグメント名，グループ名で行います．

SHORT

書式　SHORT　〈ラベル〉

セグメント内の相対分岐が＋129～－126バイト以内で，オフセットが1バイトで足りることを宣言します．相対分岐のオフセットはデフォルトで2バイトですから，メモリの節約になります．

SEG

書式　SEG　〈変数／ラベル〉

〈変数〉，〈ラベル〉の定義されているセグメントのセグメントベースを返します．これを使うことにより，細かなセグメント操作を行うことが可能になります．

OFFSET

書式　OFFSET　〈変数／ラベル〉

〈変数〉，〈ラベル〉のセグメントベースからのオフセットを返します．たとえば，変数のオフセットをレジスタに収めるときは，次のように使います．

```
0000                                code    segment
                                    assume  cs:code,ds:var
0000  BB 0000 R                     mov     bx,offset table
0003                                code    ends          ……tableのセグメントベースからのオフセット(アドレス)をBXレジスタに収める
                                    ;
0000                                var     segment
0000  30 31 32 33 34 35             table   db      '0123456789ABCDEF'
      36 37 38 39 41 42
      43 44 45 46
0010                                var     ends
                                    end
```

Fig. 6.20　OFFSET

TYPE

書式　TYPE　〈変数／ラベル〉

〈変数〉のバイト数，〈ラベル〉のNEAR，FARを返します．単純な変数の場合，それを定義した擬似命令により設定されるバイト数が返され，ストラクチャの場合は，その全バイト数が返されます．

ラベルの場合，NEAR(セグメント内)であればFFFFH，FAR(セグメント間)であればFFFEHが返されます．

LENGTH

書式　LENGTH　〈変数〉

〈変数〉がDUPによっていくつ定義されているかを返します．DUPが使われていない場合は1が返されます．この演算子は，変数の型によるバイト数に影響されません．

SIZE

書式　SIZE　〈変数〉

〈変数〉が何バイト割り当てられているかを返します．この演算子により返される値はLENGTH演算子とTYPE演算子により返される値の積となります(Fig. 6.21)．

```
0000                                code    segment
                                    assume  cs:code,ds:var      ……変数の型(バイト数)をALレジスタに収める
0000  B0 02                         mov     al,type array      ……変数の数(DUPで指定されたもの)をBXレジスタに収める
0002  BB 0064 90                    mov     bx,length array    ……変数の総バイト数をCXレジスタに収める
0006  B9 00C8 90                    mov     cx,size array
000A                                code    ends
                                    ;
0000                                var     segment
0000     64 [                       array   dw      100 dup(?)
            ????
                ]

00C8                                var     ends
                                    ;
                                            end
```

Fig. 6.21　TYPE，LENGTH，SIZE

6.5.2　一般の演算子

マクロアセンブラには，前述のアトリビュート演算子以外に，算術演算子，論理演算子，関係演算子の3種類が用意されています(Table 6.2)．

算術演算子は，オペランド間で算術演算を行い，結果を数値で返します．この場合，オペランドはかならずしも定数でなくてもかまいません．

関係演算子は，2つの定数オペランド間の大小関係を判定します．その演算子を使った関係式が真となる場合，式の値としてFFFFH，偽の場合0000Hが返されます．

論理演算子は，オペランドの各ビットごとに論理演算を行い，その結果を返します(Table 6.3)．

Table 6.2　算術演算子

演算子	書　式	機　能
+	X + Y	加算
−	X − Y	減算
*	X * Y	乗算
／	X ／ Y	除算
−	− X	単項マイナス
MOD	X　MOD　Y	X／Yの剰余
SHL	X　SHL　Y	XをYビット左シフト
SHR	X　SHR　Y	XをYビット右シフト

Table 6.3　論理演算子

演算子	書　式	機　能
NOT	NOT　X	各ビットを反転する
AND	X　AND　Y	各ビットごとに論理積をとる
OR	X　OR　Y	各ビットごとに論理和をとる
XOR	X　XOR　Y	各ビットごとに排他的論理和をとる

演算子	書　式	機　能
EQ	X　EQ　Y	X=Yのとき真を返す
NE	X　NE　Y	X≠Yのとき真を返す
LT	X　LT　Y	X<Yのとき真を返す
LE	X　LE　Y	X≦Yのとき真を返す
GT	X　GT　Y	X>Yのとき真を返す
GE	X　GE　Y	X≧Yのとき真を返す

7章　システムコール実例集

本章では，MS-DOSのユーザーに公開されているすべてのシステムコールの解説とその実行例を示します．ユーザーがアセンブラや高級言語でプログラムを記述する場合，これらのシステムコールを用いれば，プログラムを簡単にすることができるとともに，特定機種に依存しない汎用性の高いものにすることができます．

各ファンクションの解説では，コール手順，リターン情報(エラーコード，戻り値)，ファンクションの内容説明を行い，1つまたは複数のシステムコールをまとめてサンプル・プログラムを示しています．

7.1　システムコールのコール手順

システムコールは，次の3つの方法でコールすることができます．

(1) INT　21H を使う方法

- AH ←システムコールを行うファンクション番号(00H～57H)
- 他のレジスタ←個々のシステムコールで指定された値
  (このときメモリ上にも値の設定が必要な場合がある)
- INT　21H

この方法が，MS-DOSの標準的なコール方法です．

システムコールが実行されるとその処理を行い，値を返すものであれば，レジスタや指定されたメモリ領域にその値が返されます．

(2) CALL　05H を使う方法

- CL ←システムコールを行うファンクション番号(00H～24H)
- 他のレジスタ← CP／M コンパチブルのシステムコールで指定された値
- CALL　05H(コード・セグメントの 05H番地をコールする)

この方法は，CP／M との互換性を保つために用意されています．つまり，CP／M コンパチブルのシステムコール(00H～24H)にのみ利用できるものです．

(3) CALL　PSP + 50H を使う方法

- AH ←システムコールを行うファンクション番号
- 他のレジスタ←個々のシステムコールで指定された値
- CALL　PSP+50H

これは，PSP+50H のアドレスに INT　21H が書き込まれているために，FAR　CALL 命令でこのアドレスを呼べば，通常の割込みと同じ処理を行います．しかし，これはユーザーのために用意されているというよりは，MS-DOS 側のシステムコールの処理を移植性のあるものにするために用意されていると考えられます．

なお，本書のシステムコールのサンプル・プログラムは，すべて(1)の方法を採用しています．

7.2　エラーの処理

CP／M とコンパチブルなシステムコールでは，エラーが発生したか否か，またエラーの状態を返す必要があるときには AL レジスタに値を返すので，AL レジスタの値を参照することによってエラー等の判断および処理を行う必要があります．

XENIX コンパチブルのシステムコールでは，エラーが起きる可能性のあるものは，エラーが起きた場合キャリーフラグがセットされ，AX レジスタにそのエラーコードが返されます．キャリーフラグがセットされない場合には，正常終了したことを示します．

AX レジスタに返されるエラーコードは統一されていて，その意味は Table 7.1 に示すように定められています．しかし，はっきりキャリーフラグがセットされてエラーが返されるほかにも，実際には，エラーになっている場合がありますので注意が必要です．たとえば，ファンクション 40 H でディスクへ書込みを行ってディスクフルになった場合のように，キャリーフラグがセットされずに，書き込まれたバイト数が 0 となることで示される場合もあります．

Table 7.1　XENIXコンパチブルなシステムコールのエラーコード

エラーコード	エラーの意味
AX＝01H	無効なファンクション(システムコールを行う機能の設定が誤っている)
AX＝02H	該当するファイルが存在しない
AX＝03H	パス名が無効であるか，該当するパスが存在しない
AX＝04H	オープンされているファイルが多すぎる
AX＝05H	アクセス(ファイルの作成およびオープン)が否定された (不正なアクセス，たとえば，ディレクトリやボリュームラベルをアクセスした)
AX＝06H	無効なファイル・ハンドル．オープンされていないファイル・ハンドルを使用した．あるいは使用モード(R／W)の異なるファイル・ハンドルを使用した
AX＝07H	メモリ・コントロールブロックが破壊されている
AX＝08H	メモリ・アロケーションのためのメモリ領域がない
AX＝09H	無効なメモリブロックを指定した
AX＝0AH	不正な環境(環境の大きさが32Kバイト以上ある)
AX＝0BH	不正なフォーマット(EXEファイルのヘッダ部分のフォーマットが適切ではない)
AX＝0CH	無効なアクセス(ファイルのオープンでのアクセスの指定が誤っていた)
AX＝0DH	無効なデータ(システムコールを行う際に設定したデータが誤っていた)
AX＝0EH	未使用
AX＝0FH	無効なドライブ(システムコールを行う際に設定したドライブ名が誤っていた)
AX＝10H	カレント・ディレクトリを消そうとした
AX＝11H	無効なデバイス指定(デバイスの指定が限定されているシステムコールにおいて，不正なデバイスを指定した)
AX＝12H	設定したファイル名に該当するファイルがこれ以上存在しない

7.3　レジスタの保存

MS-DOSのシステムコールでは，何か値が返されるレジスタ以外は，AXレジスタを除いてすべてのレジスタは保存されます．また，エラーが生じた場合には，エラーコードやステータスを返すレジスタ以外は変化しません．したがってプログラム作成時に，システムコールで使う以外のレジスタの保存については考える必要はありません．

マクロアセンブラ(MASM)によって第Ⅲ部7章のソースリストのクロスリファレンスファイル(.CRF)の出力指定を行った場合，マクロアセンブラのバージョンによっては次のエラーメッセージが出力されます．

Not proper align／combine type (Code 25)……SEGMENTのパラメータが誤っている

このような場合には，ソースリストのセグメント指定にメモリ配置の指定を宣言することによってエラーメッセージを除去することができます(通常はPUBLIC)．なお，オブジェクトファイル(.OBJ)の生成，実行ファイル(.COM)，リスティングファイル(.LST)の作成にはまったく問題はありません．

7.4　システムコールの実行例

FUNCTION　00H, 01H, 02H

ファンクション00H　(Terminate Program)——プログラムの終了

コール手順

```
AH　←00H
CS　←PSPのセグメント・アドレス
INT　21H
```

Return

なし

解説

現在のプロセスを終了し，制御が親プロセスに移される．このとき，すべてのファイルバッファの内容はディスクに書き込まれ，すべてのオープンされているファイル・ハンドルがクローズされる．このシステムコールを行う際には，CSレジスタにPSPのセグメント・アドレスの値がはいっている必要がある．

なお，ファイルの大きさを変更したり，ファイルを作成したのであれば，このシステムコールを行う前にファイルをクローズする必要がある．そうしなければファイルの大きさがディレクトリに正しく書き込まれない．

このシステムコールは内部割込み20Hと同じ働きをするので，CSレジスタにPSPのセグメント・アドレスがはいっている場合に，JMP 0を行ってもよい．なぜなら，PSPの先頭アドレスには，INT　20Hのコード(CDH，20H)がセットされているからである．

ファンクション01H　(Read Keyboard and Echo)——キーボード入力とコンソールへのエコー

コール手順

```
AH　←01H
INT　21H
```

Return

AL←標準入力から入力された文字コード

解説

標準入力(デフォルトはキーボード)から1文字入力されるまで待ち(入力されるまで内部でループしている), 入力された文字を標準出力(デフォルトはスクリーン)にエコーバックする. もし入力された文字がCTRL-Cだった場合, 内部割込み23Hが実行される.

ファンクション02H (Display Character)——文字の出力

コール手順

AH ←02H
DL ←標準出力に出力する文字コード
INT 21H

Return

なし

解説

DLレジスタに入れた文字コードを標準出力に出力する. もし標準入力からCTRL-Cが入力された場合, 内部割込み23Hが実行される.

FUNCTION 00H, 01H, 02H Sample Program

標準入力から1文字入力されると入力された文字がエコーバックされ, ″:″に続いて入力された文字が3行分連続して標準出力に出力される. CTRL-Z(1AH)を入力するとプログラムが終了する.

```
                        ;-----------------------------------------;
                        ;   Function  00h: Terminate Program      ;
                        ;             01h: Read keyboard and Echo ;
                        ;             02h: Display Character      ;
                        ;-----------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h
                        ;
0100  B4 02             start:  mov     ah,2
0102  B2 3F                     mov     dl,'?'          "文字の出力" のシステムコール
0104  CD 21                     int     21h             "?" がスクリーンに出力される

0106  B4 01                     mov     ah,1            "キーボード入力とコンソールへ
0108  CD 21                     int     21h             のエコー"のシステムコール.
                                                        入力があるまで待つ
010A  8A D8                     mov     bl,al           入力データをBLレジスタに保存
010C  3C 1A                     cmp     al,1ah  ;=^Z    する
010E  74 25                     je      reset           CTRL-Zならば"reset"
                                                        へジャンプ

0110  B4 02                     mov     ah,2            エコーバックされた文字とプログ
0112  B2 3A                     mov     dl,':'          ラムが出力した文字を分離するた
0114  CD 21                     int     21h             め ″:″ を出力する
```

```
0116  B9 00ED                    mov    cx,237           BLレジスタに保存してある文字
                                                         を3行分(237文字)出力する
0119  B4 02           putlp:     mov    ah,2
011B  8A D3                      mov    dl,bl
011D  CD 21                      int    21h
011F  E2 F8                      loop   putlp

0121  B4 02                      mov    ah,2             CR, LF, LFを出力して次の
0123  B2 0D                      mov    dl,0dh           入力と1行あける
0125  CD 21                      int    21h
0127  B4 02                      mov    ah,2
0129  B2 0A                      mov    dl,0ah
012B  CD 21                      int    21h
012D  B4 02                      mov    ah,2
012F  B2 0A                      mov    dl,0ah
0131  CD 21                      int    21h
0133  EB CB                      jmp    start ........... "start" に戻る

0135  B4 00           reset:     mov    ah,0             "プログラムの終了" のシステム
0137  CD 21                      int    21h              コール
                      ;
0139                  cseg       ends
                      ;
                                 end    start
```

Fig. 7.1 Sample List 00H, 01H, 02H

実行手順 FUNC−00⏎

```
A>func-00⏎ ......"FUNC-00. COM"を起動する
?o:ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
 「ファンクション01Hで入力すると文字がエコーバックされる
?Z:ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
  「入力された文字が":"に続けて3行分表示される. 表示中にCTRL-S, CTRL-C等のキーが有効となる
?t:ttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
ttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
ttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt

?^Z ......CTRL-Zの入力により, MS-DOSに制御が戻る
A>■
```

Fig. 7.2 実行結果 00H, 01H, 02H

FUNCTION 03H, 04H, 05H

ファンクション 03H (Auxiliary Input)── 補助入力

コール手順

AH ←03H

INT 21H

Return

AL ←補助装置(AUX)から入力された文字コード

解説

補助装置から1文字入力されるのを待ち，入力された文字コードをALレジスタに返す．もし標準入力からCTRL-Cが入力された場合，内部割込み23Hを実行する．なお，このシステムコールはステータスやエラーコードは返さない．

ファンクション04H　(Auxiliary Output)——補助出力

コール手順

```
AH　←04H
DL　←補助装置に出力する文字コード
INT　21H
```

Return

なし

解説

DLレジスタに入れた文字コードを補助装置に出力する．このシステムコールはステータスやエラーコードは返さない．もし，標準入力からCTRL-Cが入力された場合には，内部割込み23Hが実行される．

ファンクション05H　(Print Character)——プリンタ出力

コール手順

```
AH　←05H
DL　←標準プリンタに出力する文字コード
INT　21H
```

Return

なし

解説

DLレジスタに入れた文字コードを標準プリンタに出力する．もし標準入力からCTRL-Cが入力されると内部割込み23Hが実行される．

FUNCTION　03H, 04H, 05H　　Sample Program

補助装置から1文字入力し，それを補助装置に送り返すプログラムを作成する．CTRL-P(10H)が補助装置から入力されると，それ以後の入力は補助装置だけでなく標準プリンタにも送られるようにする．もとの状態に戻すには再度CTRL-Pを入力し，CTRL-Z(1AH)が補助装置から入力されるとプログラムが終了する．

```
                                ;---------------------------------------;
                                ;   Function  03h: Auxiliary Input      ;
                                ;             04h: Auxiliary Output     ;
                                ;             05h: Print Character      ;
                                ;---------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
                                ;
0100  C6 06 013A R 00 90        start:  mov     prtflg,0 ..... prtflgを00H(プリンタに出力しない)に設定する

0106  B4 03                     chrlp:  mov     ah,3
0108  CD 21                             int     21h          } "補助入力" のシステムコール
010A  3C 1A                             cmp     al,1ah  ;=^Z
010C  74 2A                             je      return       } 入力がCTRL-Zならば "return" へジャンプする
010E  3C 10                             cmp     al,10h  ;=^P
0110  75 06                             jne     noswtch      } 入力がCTRL-Pならば prtflgを反転する
0112  F6 16 013A R                      not     prtflg
0116  EB EE                             jmp     chrlp ..... "chrlp" に戻る

0118  3C 0D                     noswtch:cmp     al,0dh
011A  75 05                             jne     nocrlf       } 入力がCRならCR，LFを出力する
011C  E8 0126 R                         call    chrout
011F  B0 0A                             mov     al,0ah

0121  E8 0126 R                 nocrlf: call    chrout       } その他の入力なら，入力した文字をそのまま出力して "補助入力" に戻る
0124  EB E0                             jmp     chrlp

0126  8A D0                     chrout: mov     dl,al
0128  B4 04                             mov     ah,4         } ALレジスタの値を補助装置に出力する
012A  CD 21                             int     21h
012C  80 3E 013A R 00                   cmp     prtflg,0
0131  74 04                             je      noprt        } "prtflg" が00Hでなければ，ALレジスタの値を標準プリンタに出力する
0133  B4 05                             mov     ah,5
0135  CD 21                             int     21h
0137  C3                        noprt:  ret ..... リターン

0138  CD 20                     return: int     20h ..... プログラムの終了
                                ;
013A  ??                        prtflg  db      ?
                                ;
013B                            cseg    ends
                                ;
                                        end     start
```

Fig. 7.3　Sample List 03H, 04H, 05H

実行手順

まず，補助装置との通信に必要な設定を行う．

例：PC-9801 と PC-8201 を RS-232C で接続する場合

PC-9801 側　SPEED　RS232C-0　9600　BITS-8　PARITY-NONE

STOP-1　XON ↵

PC-8201 側　TELCOM を起動する

STAT　8N81XN ↵

TERM ↵

通信形式は 9600bps, パリティなし，8 ビット長，1 ストップビット，X パラメータによる制御を行うものとする．

設定後，プログラムを起動する．

FUNC−03 ↵

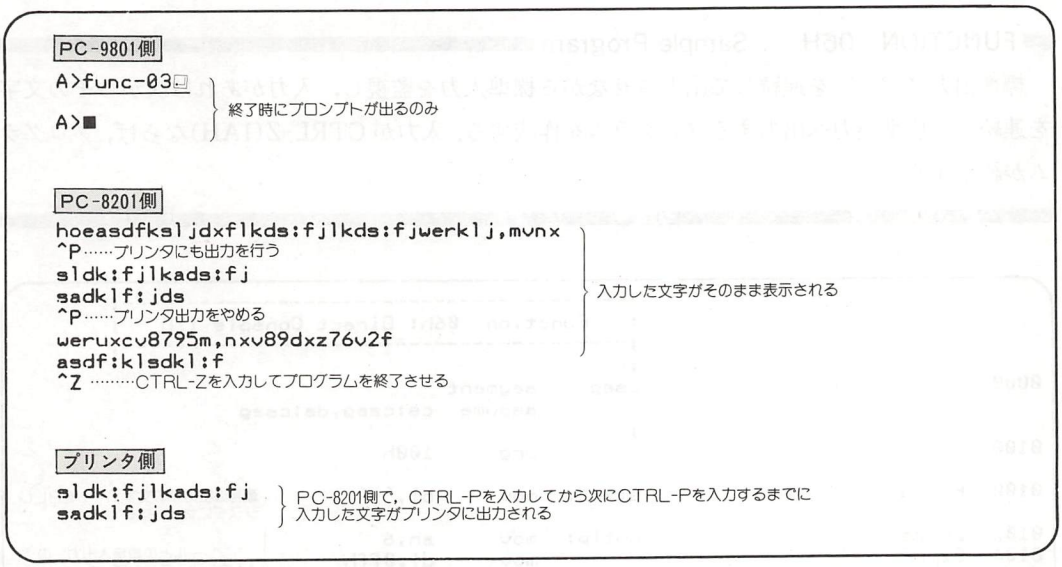


Fig. 7．4　実行結果 03H，04H，05H

FUNCTION　06H

ファンクション 06H　(Direct Console I／O)——コンソールとの直接入出力

コール手順

文字を出力する場合

AH　← 06H

DL　← FFH 以外の値

INT　21H

文字を入力する場合

AH　← 06H

DL　← FFH

INT　21H

Return

文字を出力する場合

なし

文字を入力する場合

ゼロフラグがセットされている場合

AL ← 00H(入力なし)

ゼロフラグがセットされていない場合

AL ←入力した文字コード

解説

標準入力から文字コードを入力したり，標準出力に文字コードを出力したりする．このシステムコールではブレークチェック(CTRL-C の判定)は行わない．

FUNCTION　06H　　Sample Program

標準出力に〝—〞を連続して出力させながら標準入力を監視し，入力があれば次からその文字を連続して標準出力へ出力するプログラムを作成する．入力が CTRL-Z(1AH)ならば，プログラムが終了する．

```
                        ;---------------------------------------;
                        ;   Function  06h: Direct Console I/O   ;
                        ;---------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h

0100  B3 2D             start:  mov     bl,'-'          最初に出力する文字〝-〞をBLレジスタにセットする

0102  B4 06             outlp:  mov     ah,6            〝コンソールとの直接入出力〞の
0104  B2 FF                     mov     dl,0ffh         〝入力〞のシステムコール
0106  CD 21                     int     21h
0108  74 06                     jz      chrout          入力がなければ(ゼロフラグがセット)〝chrout〞へジャンプする

010A  3C 1A                     cmp     al,1ah  ;=^Z    入力がCTRL-Zなら〝return〞へ
010C  74 0A                     je      return          ジャンプする
010E  8A D8                     mov     bl,al           ALレジスタの値をこれ以後出力する文字としてBLレジスタに入れる

0110  B4 06             chrout: mov     ah,6            〝コンソールとの直接入出力〞の
0112  8A D3                     mov     dl,bl           〝出力〞のシステムコール
0114  CD 21                     int     21h
0116  EB EA                     jmp     outlp           〝outlp〞に戻る

0118  CD 20             return: int     20h             プログラムの終了
                        ;
011A                    cseg    ends
                        ;
                                end     start
```

Fig. 7 . 5　Sample List　0 6 H

実行手順　　FUNC－06 ↵

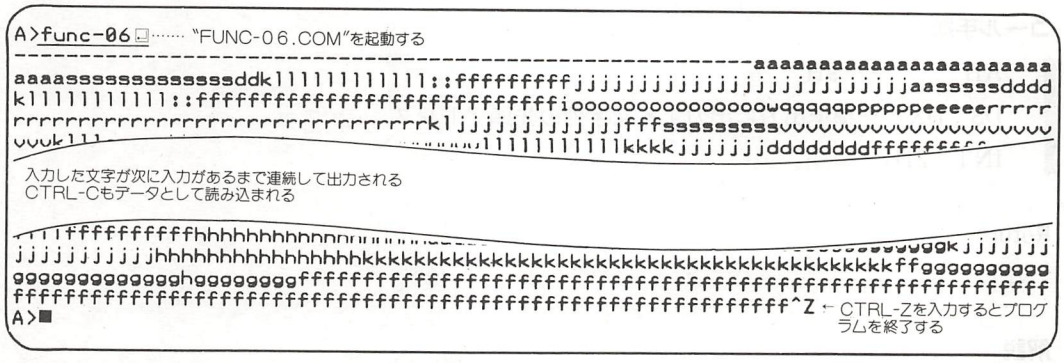


Fig. 7．6　実行結果　06H

FUNCTION　07H, 08H, 09H

ファンクション 07H　(Direct Console Input)——コンソールからの直接入力

コール手順

```
AH　←07H
INT　21H
```

Return

AL←標準入力から入力された文字コード

解説

1文字入力されるまで待ち，入力された文字コードをALレジスタに返す．文字のエコーおよびブレークチェックは行わない．

ファンクション 08H　(Read Keyboard)——エコーなしのキーボード入力

コール手順

```
AH　←08H
INT　21H
```

Return

AL←標準入力から入力された文字コード

解説

1文字入力されるまで待ち，入力された文字コードをALレジスタに返す．

文字のエコーは行わないがブレークチェックを行う(ファンクション07Hとこの点が異なる)．したがって，入力待ちのときCTRL-Cが入力されると内部割込み23Hが実行される．

ファンクション 09H　(Display String)——文字列の出力

コール手順

AH　　← 09H
DS：DX　←標準出力に出力する文字列の先頭アドレス
INT　21H

Return

なし

解説

文字列の格納されている先頭アドレスのセグメントを DS レジスタ，オフセットを DX レジスタに入れて，システムコールを行う．文字列の最後は〝$〟でなければならず，文字列の最後を示す〝$〟の直前まで標準出力に出力され，〝$〟は出力されない．

文字列出力中に CTRL-C が入力されると内部割込み 23H が実行される．

FUNCTION　07H, 08H, 09H　　Sample Program

エコーバックをしないで文字を読み込み，入力された文字列のダンプ出力を行うプログラムを作成する．ファンクション 07H と 08H の入力を区別するため，プロンプトをファンクション 09H を用いて表示している．プロンプトの後に 16 文字ずつ入力すると，ファンクション 07H による入力では，CTRL-C もデータとして入力されるが，ファンクション 08H による入力では，プログラムが終了する．

```
                        ;-----------------------------------------;
                        ;   Function  07h: Direct Console Input   ;
                        ;             08h: Read Keyboard          ;
                        ;             09h: Display String         ;
                        ;-----------------------------------------;
                        ;
0000                    cseg    segment
                                extrn   crlfout:near,dump:near
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h
                        ;
0100  B4 09             start:  mov     ah,9
0102  BA 0146 R                 mov     dx,offset prompt1
0105  CD 21                     int     21h
0107  BB 0166 R                 mov     bx,offset buf1
010A  B9 0010                   mov     cx,16
010D  B4 07             getlp1: mov     ah,7
010F  CD 21                     int     21h
0111  88 07                     mov     [bx],al
0113  43                        inc     bx
0114  E2 F7                     loop    getlp1
```

〝crlfout〟と〝dump〟を外部サブルーチンとして定義する

〝文字列の出力〟のシステムコール
〝prompt 1〟の文字列が出力される

入力データを格納する〝buf 1〟のアドレスをBXレジスタに，バイト長をCXレジスタにセットする

〝コンソールからの直接入力〟のシステムコール
文字のエコー，ブレークチェックは行わない

〝buf 1〟がいっぱいになるまで入力された文字を格納していく

```
0116  E8 0000 E                        call    crlfout
0119  BB 0166 R                        mov     bx,offset buf1   ; "buf 1" の16バイトをダンプ形式で出力する
011C  B9 0010                          mov     cx,16
011F  E8 0000 E                        call    dump

0122  B4 09                            mov     ah,9             ; "文字列の出力" のシステムコール
0124  BA 0156 R                        mov     dx,offset prompt2 ; "prompt2"の文字列が出力される
0127  CD 21                            int     21h

0129  BB 0176 R                        mov     bx,offset buf2   ; 入力データを格納する"buf 2"のアドレスをBXレジスタに,
012C  B9 0010                          mov     cx,16            ; バイト長をCXレジスタにセットする

012F  B4 08                  getlp2:   mov     ah,8             ; "エコーなしのキーボード入力" のシステムコール
0131  CD 21                            int     21h              ; 文字のエコーは行わないがブレークチェックは行う
0133  88 07                            mov     [bx],al          ; "buf 2" がいっぱいになるまで入力された
0135  43                               inc     bx               ; 文字を格納していく
0136  E2 F7                            loop    getlp2

0138  E8 0000 E                        call    crlfout
013B  BB 0176 R                        mov     bx,offset buf2   ; "buf 2" の16バイトをダンプ形式で出力する
013E  B9 0010                          mov     cx,16
0141  E8 0000 E                        call    dump

0144  CD 20                            int     20h              ; プログラムの終了
                             ;
0146  0D 0A 46 75 6E 63      prompt1 db  0dh,0ah,'Function 7 ? $'   ; 出力する文字列
      74 69 6F 6E 20 37
      20 3F 20 24
0156  0D 0A 46 75 6E 63      prompt2 db  0dh,0ah,'Function 8 ? $'
      74 69 6F 6E 20 38
      20 3F 20 24
0166     10 [                buf1    db      16 dup (?)
            ??
               ]

0176     10 [                buf2    db      16 dup (?)
            ??
               ]

                             ;
0186                         cseg    ends
                             ;
                                     end     start
```

Fig. 7.7　Sample List 07H, 08H, 09H

実行手順　　FUNC−07 ↵

```
A>func-07 ↵   ……"FUNC-07.COM" を起動する

Function 7 ? ……ファンクション07Hではエコーバックは行われない
73 64 61 66 69 6F 75 77 38 39 33 35 34 6B 6A 64    sdafiouw89354kjd

Function 8 ? ……ファンクション08Hも同様
73 66 69 6F 75 76 6A 6E 73 66 6A 66 64 73 61 6B    sfiouvjnsfjfdsak

A>func-07 ↵   ……再び"FUNC-07.COM" を起動する

Function 7 ? ……ファンクション07HではCTRL-Cもデータとして入力される
13 02 6B 6C 6A 66 73 68 03 03 03 03 03 03 03 03,   ..kljfsh........
                          └──CTRL-Cがデーターとして取り込まれている
Function 8 ?^C ……ファンクション08HではCTRL-Cを入力すると
                   プログラムが終了してしまう
A>■
```

Fig. 7.8　実行結果　07H, 08H, 09H

●外部参照のサブルーチン

メモリの内容やレジスタの値は，プログラムをDEBUGで実行させながら調べていってもよいのですが，本章では，メモリをダンプしたりレジスタの値を表示させるサブルーチン(“DUMP．ASM”)をあらかじめ作成しておいて，サンプル・プログラムとリンクさせて用いています．プログラムでこれらのサブルーチンを用いるにはあらかじめ

extrn　〈サブルーチン名〉: near, ……, 〈サブルーチン名〉: near

という文を挿入しておいてアセンブルしておき，“DUMP．ASM”をアセンブルして作成した“DUMP．OBJ”とリンクさせます．この中の

dump,　hexout,　crlfout,　putch

の4つのサブルーチンの内のいくつかをextrnで宣言しているときには，この“DUMP．OBJ”をリンクして初めて実際に動作するプログラムとなることに気を付けてください．Fig. 7.9にこの“DUMP．ASM”のLST形式のファイルを示します．

```
                                ;---------------------------------------;
                                ;       Dump CX bytes from DS:BX        ;
                                ;---------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                        public  dump,hexout,crlfout,putch
                                ;               └PUBLICで宣言して他のモジュールから外部参照できるようにする
                                ;------ dump ----------;  (DS：BXレジスタで与えられたアドレスからCXバイトをダンプ形式で標準出力へ出力するサブルーチン)
                                ;
0000                            dump    proc    near
                                ;
0000  50                                push    ax      }
0001  53                                push    bx      } レジスタの退避
0002  51                                push    cx      }

0003  83 C1 0F                          add     cx,15   }
0006  D1 E9                             shr     cx,1    } CXレジスタの値をパラグラフ数に変換する
0008  D1 E9                             shr     cx,1    }   CX＊(CX＋15)¥16
000A  D1 E9                             shr     cx,1    }
000C  D1 E9                             shr     cx,1    }
000E  E3 40                             jcxz    next    } CXレジスタが0000Hならば“next”へジャンプする

0010  51                        linelp: push    cx      ……レジスタの退避

0011  B9 0010                           mov     cx,10h  ……出力するバイト数(16バイト)
0014  53                                push    bx      ……表示を開始するオフセットを退避する

0015  8A 07                     hexlp:  mov     al,[bx] }
0017  E8 005F R                         call    hexdisp } DS：BXレジスタで指定されたアドレスから16バイトを
001A  B0 20                             mov     al,' '  }   □□ □□……□□
001C  E8 008B R                         call    putch   } の形式で標準出力に出力する
001F  43                                inc     bx      }
0020  E2 F3                             loop    hexlp   }

0022  B0 20                             mov     al,' '  }
0024  E8 008B R                         call    putch   } スペースを2つ標準出力に出力する
0027  B0 20                             mov     al,' '  }
0029  E8 008B R                         call    putch   }
```

```
002C  5B                              pop     bx ……表示を開始するオフセットの値を復帰する
002D  B9 0010                         mov     cx,10h……出力するバイト数（16バイト）

0030  8A 07                   chrlp:  mov     al,[bx]
0032  3C 20                           cmp     al,' '
0034  72 0C                           jb      badchr
0036  3C 7F                           cmp     al,7fh
0038  72 0A                           jb      goodchr
003A  3C A0                           cmp     al,0a0h   DS：BXレジスタで指定されたアドレスから16バイトをキャラクタコードで標準出力に出力する．このときコントロールコード，グラフィックキャラクタは"."に変換される
003C  72 04                           jb      badchr
003E  3C E0                           cmp     al,0e0h
0040  72 02                           jb      goodchr
0042  B0 2E                   badchr: mov     al,'.'
0044  E8 008B R               goodchr:call    putch
0047  43                              inc     bx
0048  E2 E6                           loop    chrlp

004A  E8 007E R                       call    crlfout …… 1行改行する

004D  59                              pop     cx        指定されたパラグラフ数を出力するまで"linelp"に戻る
004E  E2 C0                           loop    linelp

0050  59                      next:   pop     cx
0051  5B                              pop     bx        レジスタの値を復帰させる
0052  58                              pop     ax
0053  C3                              ret ……リターン
                              ;
0054                          dump    endp
                              ;
                              ;------ hexout ---------;（AXレジスタの値を16進4桁で標準出力に出力するサブルーチン）
                              ;
0054                          hexout  proc    near
                              ;
0054  50                              push    ax ……レジスタを退避させる
0055  8A C4                           mov     al,ah     AHレジスタの値を16進2桁で標準出力へ出力する
0057  E8 005F R                       call    hexdisp
005A  58                              pop     ax ……レジスタの値を復帰させる
005B  E8 005F R                       call    hexdisp…ALレジスタの値を16進2桁で標準出力へ出力する
005E  C3                              ret ……リターン
                              ;
005F                          hexout  endp
                              ;
                              ;------ hexdisp --------;（ALレジスタの値を16進2桁で標準出力に出力するサブルーチン）
                              ;
005F                          hexdisp proc    near
                              ;
005F  50                              push    ax        レジスタの退避
0060  51                              push    cx
0061  B1 04                           mov     cl,4      上位4ビットを16進1桁で標準出力へ出力する
0063  D2 C8                           ror     al,cl
0065  E8 006E R                       call    half
0068  59                              pop     cx        レジスタを復帰させる
0069  58                              pop     ax
006A  E8 006E R                       call    half ……下位4ビットを16進1桁で標準出力へ出力する
006D  C3                              ret ……リターン
                              ;
006E                          hexdisp endp
                              ;
                              ;------ half -----------;（ALレジスタの下位4ビットを16進1桁で標準出力に出力するサブルーチン）
                              ;
006E                          half    proc    near
                              ;
006E  50                              push    ax ……レジスタを退避させる
006F  24 0F                           and     al,0fh……ALレジスタの上位4ビットをマスクする
0071  04 30                           add     al,30h    ALレジスタの下位4ビットを16進1桁の文字コードに変換して標準出力に出力する
0073  3C 3A                           cmp     al,3ah
0075  72 02                           jb      half1
0077  04 07                           add     al,7
0079  E8 008B R              half1:   call    putch
007C  58                              pop     ax……レジスタの値を復帰させる
007D  C3                              ret ……リターン
007E                          half    endp
```

```
                                ;
                                ;------ crlfout --------;  (CR, LFを標準出力に出力するサブルーチン)
                                ;
007E                            crlfout proc    near
                                ;
007E  50                                push    ax ..... レジスタを退避させる
007F  B0 0D                             mov     al,0dh  }
0081  E8 008B R                         call    putch   } CRを標準出力に出力する
0084  B0 0A                             mov     al,0ah  }
0086  E8 008B R                         call    putch   } LFを標準出力に出力する
0089  58                                pop     ax ..... レジスタの値を復帰させる
008A  C3                                ret ..... リターン
                                ;
008B                            crlfout endp
                                ;
                                ;------ putch ----------;  (ALレジスタの文字コードを標準出力に出力するサブルーチン)
                                ;
008B                            putch   proc    near
                                ;
008B  50                                push    ax      }
008C  52                                push    dx      } レジスタを退避させる
008D  B4 02                             mov     ah,2    }
008F  8A D0                             mov     dl,al   } "文字の出力" のシステムコール
0091  CD 21                             int     21h     }
0093  5A                                pop     dx      }
0094  58                                pop     ax      } レジスタの値を復帰させる
0095  C3                                ret ..... リターン
                                ;
0096                            putch   endp
                                ;
0096                            cseg    ends
                                ;
                                        end
```

Fig. 7.9　DUMP.LST　07H, 08H, 09H

FUNCTION　0AH

ファンクション0AH　(Buffered Keyboard Input)——バッファードキーボード入力

コール手順

AH　　←0AH
DS：DX　←入力バッファの先頭アドレス
INT　21H

Return

なし

解説

標準入力からの入力を，〝入力バッファの先頭アドレス＋2〟番地から順次格納し，リターンが入力されると，システムコールから戻る．入力のときには，コマンドラインと同様に各種のテンプレート機能が使用できる．入力バッファが最後のバイトの1つ前まで埋められると，それ以後に入力された文字は無視され，アスキーコードのBEL(07H)をリターンキーが押されるまで標準出力に出力する(Fig. 7.10)．

DS:DXで示される入力バッファの先頭アドレス

mx	nc	C_1	C_2	C_3	·····	Cnc	CR

mx…入力許可最大文字数（00H～FFH）をセットする．mx＝0のときは何もせずにプログラムに戻る
nc…システムコールから戻ったとき，実際に入力された文字数がセットされる(ただし，キャリッジリターンを入れない文字数)
C_1…文字
CR…キャリッジリターン

Fig. 7.10　入力バッファ

テンプレート機能を用いる際に，入力バッファをこのファンクションによって返される形式でセットしておけば，それがテンプレートの内容となる．したがって，入力バッファを破壊せずに毎回用いれば，コマンドラインと同じような感覚でプログラム中でも入力できる．

CTRL-Cが入力されると，内部割込み23Hが実行される．

FUNCTION　0AH　　Sample Program

リターン(0DH)を加えた入力許可文字数を46文字とし，文字列の入力を行うプログラムを作成する．

プログラムを実行すると，まずプロンプト〝*〟が出力され，プロンプトの後ろに続けて適当な文字列をキー入力してリターンキーを押すと，コンソールバッファの内容がダンプ形式で表示され，再びプロンプトを出力して入力待ちとなる．プログラムの終了には，CTRL-Cを入力する．

```
                        ;-----------------------------------------------;
                        ;   Function  0ah: Buffered Keyboard Input      ;
                        ;-----------------------------------------------;
                        ;
0000                    cseg    segment
                                extrn   crlfout:near,putch:near
                                extrn   dump:near
                                assume  cs:cseg,ds:cseg
                        ;               ;
0100                            org     100h
                        ;
0100  E8 0000 E         start:  call    crlfout
0103  E8 0000 E                call    crlfout

0106  B0 2A                     mov     al,'*'
0108  E8 0000 E                call    putch

010B  B4 0A                     mov     ah,0ah
010D  BA 0123 R                 mov     dx,offset buffer
0110  CD 21                     int     21h

0112  E8 0000 E                call    crlfout
0115  E8 0000 E                call    crlfout

0118  BB 0123 R                 mov     bx,offset buffer
011B  B9 0030                   mov     cx,48
011E  E8 0000 E                 call    dump

0121  EB DD                     jmp     start
                        ;
```

"crlfout","putch","dump"を外部サブルーチンとして定義する

2行改行する

"*"を標準出力に出力する

"バッファード・キーボード入力"のシステムコール
入力中にはテンプレート機能が活用できる

2行改行する

"buffer"内の48バイトをダンプ形式で出力する

"start"に戻る

```
0123  2E 2F [                    buffer  db      46,47 dup (?)
               ??
                  ]

                                  ;
0153                              cseg    ends
                                  ;
                                          end     start
```

Fig. 7.11　Sample List　0AH

実行手順　　FUNC－0A ↵

```
A>func-0a ↵……"FUNC-0A.COM" を起動

*System call sample ↵ ……左の文字列をキー入力
      入力バッファの最大入力文字数46文字のうち18文字入力されている
2E 12 53 79 73 74 65 6D 20 63 61 6C 6C 20 73 61   ..System call sa
6D 70 6C 65 0D 00 00 00 00 00 00 00 00 00 00 00   mple............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
                                                  } 入力したデータはこのように蓄えられる

*System call 0AH sample ↵ ……"CU _ CU _ INS _ 0AH CA "と入力し、テンプレート機能を利用して"0AH"を挿入する

2E 16 53 79 73 74 65 6D 20 63 61 6C 6C 20 30 41   ..System call 0A
48 20 73 61 6D 70 6C 65 0D 00 00 00 00 00 00 00   H sample........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................

          "CA" と入力しここまで表示させる
*System call 0AH sample -- Buffered keyboard i ↵ ……"i"まで入力したところでバッファがいっぱいとなり
    入力バッファの最大入力文字数－1までしか文字を入力できない          それ以上は入力できない
2E 2D 53 79 73 74 65 6D 20 63 61 6C 6C 20 30 41   .-System call 0A
48 20 73 61 6D 70 6C 65 20 2D 2D 20 42 75 66 66   H sample -- Buff
65 72 65 64 20 6B 65 79 62 6F 61 72 64 20 69 0D   ered keyboard i.

*Buffered keyboard input ↵ ……"SUBCA"と入力し，"B" の前をスキップして残りを表示させて "i" の続きを入力する
      この部分のデータが有効
2E 17 42 75 66 66 65 72 65 64 20 6B 65 79 62 6F   ..Buffered keybo
61 72 64 20 69 6E 70 75 74 0D 2D 20 42 75 66 66   ard input.- Buff
65 72 65 64 20 6B 65 79 62 6F 61 72 64 20 69 0D   ered keyboard i.
                                  バッファの残りは前のデータがそのまま残っている

*^C ……CTRL-Cを入力すると内部割込み23Hが起こりプログラムが終了する(入力中ならいつでも有効)

A>■
```

Fig. 7.12　実行結果　0AH

FUNCTION　0BH, 2AH, 2BH, 2CH, 2DH

ファンクション0BH　(Check Keyboard Status)

——キーボードのステータスチェック

コール手順

AH　←0BH

INT　21H

Return

AL＝00H　タイプアヘッドバッファは空である

AL＝FFH　タイプアヘッドバッファに文字がはいっている

解説

タイプアヘッドバッファ内に文字がはいっているかどうかをチェックする．もしCTRL-Cがバッファ内にはいっている場合には，内部割込み23Hが実行される．

ファンクション2AH　(Get Date)——日付の読出し

コール手順

```
AH　←2AH
INT　21H
```

Return

CX←年(1980～2079)

DH←月(1～12)

DL←日(1～31)

AL←曜日(0：日，1：月，……6：土)

解説

CX，DXおよびALレジスタに現在の日付と曜日が2進数で返される．

ファンクション2BH　(Set Date)——日付の設定

コール手順

```
AH　←2BH
CX　←年(1980～2079)
DH　←月(1～12)
DL　←日(1～31)
INT　21H
```

Return

AL＝00H　日付がセットされた

AL＝FFH　日付は無効である

解説

CX，DXレジスタに2進数で与えられた年，月，日がセットされ，曜日は自動的に計算される．

ファンクション 2CH　(Get Time)——時刻の読出し

コール手順

```
AH　←2CH
INT　21H
```

Return

CH ←時(0～23)

CL ←分(0～59)

DH ←秒(0～59)

DL ←1／100 秒(0～99)　(ハードウェアで秒単位でしか計測できない場合にはつねに 00H)

解説

現在の時刻を 2 進数で CX，DX レジスタに返す．

ファンクション 2DH　(Set Time)—— 時刻の設定

コール手順

```
AH　←2DH
CH　←時(0～23)
CL　←分(0～59)
DH　←秒(0～59)
DL　←1／100 秒(0～99)　(ハードウェアで 100 分の 1 秒まで設定できない場合は無効だ
　　が，値は 0～99 の範囲になければならない)
INT　21H
```

Return

AL＝00H　時刻がセットされた

AL＝FFH　時刻が無効である

解説

CX，DX レジスタに 2 進数で与えられた時刻をセットする．

FUNCTION　0BH, 2AH, 2BH, 2CH, 2DH　Sample Program

年，月，日を設定して，その曜日を表示し，時刻を 00：00：00 にセットしてキーボードから 1 文字入力されるまで時刻を表示し続けるプログラムを作成する．

```
                        ;-----------------------------------------;
                        ;   Function  0bh: Check Keyboard Status  ;
                        ;             2ah: Get Date               ;
                        ;             2bh: Set Date               ;
                        ;             2ch: Get Time               ;
                        ;             2dh: Set Time               ;
                        ;-----------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h
                        ;
0100  B4 09             start:  mov     ah,9            ┐"年/月/日 is" と出力
0102  BA 016E R                 mov     dx,offset msgday│する. このプログラムで
0105  CD 21                     int     21h             ┘は2001年1月1日と
                                                        しておく
0107  B4 2B                     mov     ah,2bh          ┐"日付の設定" のシス
0109  B9 07D1                   mov     cx,2001         │テムコール
010C  B6 01                     mov     dh,1            │"2001年1月1日"に
010E  B2 01                     mov     dl,1            │日付をセットする
0110  CD 21                     int     21h             ┘

0112  B4 2A                     mov     ah,2ah          ┐"日付の読出し"のシステ
0114  CD 21                     int     21h             ┘ムコールALレジスタに
                                                        曜日が返される
0116  BA 017B R                 mov     dx,offset day   ┐
0119  32 E4                     xor     ah,ah           │
011B  D1 E0                     shl     ax,1            │
011D  D1 E0                     shl     ax,1            │ALレジスタの曜日を,
011F  03 D0                     add     dx,ax           │曜日のテーブル "day"
0121  B4 09                     mov     ah,9            │から出力する
0123  CD 21                     int     21h             │
0125  BA 0197 R                 mov     dx,offset crlf  │
0128  CD 21                     int     21h             ┘

012A  B4 2D                     mov     ah,2dh          ┐
012C  B5 00                     mov     ch,0            │"時刻の設定" のシステ
012E  B1 00                     mov     cl,0            │ムコール
0130  B6 00                     mov     dh,0            │"00:00:00.00"
0132  B2 00                     mov     dl,0            │に時刻をセットする
0134  CD 21                     int     21h             ┘

0136  B4 2C             timelp: mov     ah,2ch          ┐"時刻の読出し"のシス
0138  CD 21                     int     21h             ┘テムコール

013A  BB 019A R                 mov     bx,offset time[0]┐
013D  8A C5                     mov     al,ch           │
013F  E8 0163 R                 call    convert         │
0142  BB 019D R                 mov     bx,offset time[3]│
0145  8A C1                     mov     al,cl           │
0147  E8 0163 R                 call    convert         │時刻を秒単位で表示する
014A  BB 01A0 R                 mov     bx,offset time[6]│
014D  8A C6                     mov     al,dh           │
014F  E8 0163 R                 call    convert         │
0152  B4 09                     mov     ah,9            │
0154  BA 019A R                 mov     dx,offset time  │
0157  CD 21                     int     21h             ┘

0159  B4 0B                     mov     ah,0bh          ┐
015B  CD 21                     int     21h             │キー入力がなければ
015D  3C FF                     cmp     al,0ffh         │"timelp" へ戻って時刻
015F  75 D5                     jne     timelp          ┘の表示を続ける

0161  CD 20                     int     20h ..... キー入力があればプログラムを終了する

0163  D4 0A             convert:aam
0165  0D 3030                   or      ax,3030h        ┐
0168  88 27                     mov     [bx],ah         │ALレジスタの値を[BX], [BX]+1
016A  88 47 01                  mov     [bx]+1,al       │に10進2桁でセットするサブルーチン
016D  C3                        ret                     ┘
```

```
                                        ;
016E  32 30 30 31 2F 31                 msgday  db  '2001/1/1 is $'
      2F 31 20 69 73 20
      24
017B  53 75 6E 24 4D 6F                 day     db  'Sun$Mon$Tue$Wed$Thu$Fri$Sat$'
      6E 24 54 75 65 24
      57 65 64 24 54 68
      75 24 46 72 69 24
      53 61 74 24
0197  0D 0A 24                           crlf    db  0dh,0ah,'$'
019A  30 30 3A 30 30 3A                 time    db  '00:00:00',0dh,'$'
      30 30 0D 24
                                        ;
01A4                                    cseg    ends
                                        ;
                                                end     start
```

Fig. 7.13　Sample List　0BH, 2AH, 2BH, 2CH, 2DH

実行手順　　FUNC−0B↵

```
A>func-0b↵ ……"FUNC-0B.COM" を実行する
2001/1/1 is Mon ……2001年1月1日は月曜日と表示される
00:00:11 ……00：00：00から何かキーを押すまで時間を表示する
A>■
```

Fig. 7.14　実行結果　0BH, 2AH, 2BH, 2CH, 2DH

FUNCTION　0CH

ファンクション0CH　(Flush Buffer, Read Keyboard)
——バッファを空にしてから入力

コール手順

AH　←0CH
AL　←01H, 06H, 07H, 08H, 0AH(このALレジスタの値に対応するファンクション番号のシステムコールが実行される)
AL　←それ以外の値(タイプアヘッドバッファを空にする)
INT　21H

Return

ALレジスタが入力用のファンクション番号を指定していない場合には，タイプアヘッドバッファを空にしてALレジスタに00Hを返す．

解説

ALレジスタに01H, 06H, 07H, 08H, 0AHの値がはいっていた場合，まずタイプアヘッドバッファを空にしてから対応するMS-DOSのシステムコールが実行される．

このファンクションはタイプアヘッドバッファを空にしてALレジスタの値によってファンクションを実行するのだから，以下の2つのコール手順は同じ意味を持つ．

```
MOV   AH , 0CH
MOV   AL ,〈実行させたい入力用ファンクション番号〉
INT   21H
```

```
MOV   AH , 0CH
MOV   AL , 00H(01H, 06H, 07H, 08H, 0AH 以外なら何でもよい)
INT   21H
MOV   AH ,〈対応するファンクション番号〉
INT   21H
```

いままでのシステムコールのサンプルをファンクション0CHを用いて書き換えてタイプアヘッドバッファが空にされることを試してみてください．

FUNCTION 0DH, 0EH, 11H, 12H

ファンクション0DH (Disk Reset)——ディスクのリセット

コール手順

```
AH   ←0DH
INT  21H
```

Return

なし

解説

このシステムコールはすべてのファイルバッファをクリアするが，ディレクトリ・エントリの更新は行わないので，ディレクトリ・エントリの更新を行うためには変更されたファイルをクローズしなければならない．ブレークチェックがONであればディスクの書込み中でもCTRL-Cが有効なので，CTRL-Cの割込み処理ルーチンではこのシステムコールを行う必要がある．

CP／Mでは一度ログインしたり，アクセスした場合には，ディスクを交換するとリードオンリーになってしまうが，MS-DOSでは，読出し／書込みが可能であるので，ファイルがクローズしてあれば，ディスクを交換するようなプログラムもこのシステムコールを行う必要はない．

ファンクション0EH　(Select Disk)——ドライブの選択

コール手順

```
AH　←0EH
DL　←ドライブ番号(0：A, 1：B, ……)
INT　21H
```

Return

AL←接続されているドライブの数

解説

DLレジスタで指定されたドライブが，カレント・ドライブとして選択され，接続されているドライブの数がALレジスタに返される．無効なドライブを選択した場合，カレント・ドライブは変更されない．

ファンクション11H　(Search For First Entry)——最初のディレクトリ・エントリの検索

コール手順

```
AH　　←11H
DS：DX　←オープンされていないFCBの先頭アドレス
INT　21H
```

Return

AL＝00H　ディレクトリ・エントリが存在する
AL＝FFH　ディレクトリ・エントリが存在しない

解説

カレント・ディレクトリを検索して，FCBのファイル名(ワイルドカードを含む)に一致するディレクトリ・エントリが存在すれば，そのディレクトリ情報をFCBと同じ形式でDTAにセットする．このときDS：DXはオープンされていないFCBの先頭アドレスを指していなければならない．ただし隠されたファイルあるいはシステムファイルを検索する場合には，DXレジスタは拡張FCBの先頭バイト(FFHが格納されているアドレス)を指している必要がある．

DTAに返されるディレクトリ情報は，通常のFCBであれば通常のFCB(37バイト)，拡張FCBならば拡張FCB(44バイト)と同じ形式のデータである．ただしワイルドカードを用いた検索では，実際のファイル名がセットされ，FCBのオフセット0CHにはファイルの属性，FCBのオフセット21H(相対レコード・フィールド)にはファイルの大きさ(バイト長)がセットされる．

ファンクション12H (Search For Next Entry)
——次のディレクトリ・エントリの検索

コール手順

```
AH      ←12H
DS：DX  ←オープンされていないFCBの先頭アドレス
INT  21H
```

Return

AL＝00H　ディレクトリ・エントリが存在する
AL＝FFH　ディレクトリ・エントリが存在しない

解説

DS：DXレジスタは，前にファンクション11Hを行ったときに指定したFCBの先頭アドレスを示していなければならない．ファンクション11Hの検索で名前の一致したディレクトリ・エントリから後の部分のディレクトリを対象にファイルを捜す．FCBのファイル名と一致するファイルが存在すれば，ファンクション11Hと同様にDTA内にディレクトリ情報が返される．

FUNCTION　0DH, 0EH, 11H, 12H　　Sample Program

カレント・ドライブをドライブAに設定し(ファンクション0EH)，指定されたファイル名を検索してファイルの情報を表示するプログラムを作成する．ドライブ名を指定しなければ，つねにドライブAのファイルが検索される．

```
                        ;-----------------------------------------------;
                        ;   Function  0dh: Disk Reset                   ;
                        ;             0eh: Select Disk                  ;
                        ;             11h: Search For First Entry       ;
                        ;             12h: Search For Next Entry        ;
                        ;-----------------------------------------------;
                        ;
0000                    cseg    segment
                                extrn   dump:near,crlfout:near
                                assume  cs:cseg,ds:cseg
                        ;
005C                            org     5ch
                        ;
005C    24 [            fcb     db      36 dup (?)
            ??
                ]
                        ;
0080    80 [            dta     db      128 dup (?)
            ??
                ]
                        ;
0100                            org     100h
                        ;
```

"dump"と"crlfout"を外部サブルーチンとして定義する

```
0100  B4 0D                    start:  mov     ah,0dh      } "ディスク・リセット" のシステムコール
0102  CD 21                            int     21h         }

0104  B4 0E                            mov     ah,0eh      } "ドライブの選択" のシステムコール
0106  B2 00                            mov     dl,0        } ドライブA(DL=00H)をカレン
0108  CD 21                            int     21h         } ト・ドライブとする

010A  B4 11                            mov     ah,11h      } "最初のディレクトリ・エントリの
010C  BA 005C R                        mov     dx,offset fcb } 検索" のシステムコール. FCBは
010F  CD 21                            int     21h         } PSP内のデフォルトFCBを使用
0111  3C FF                            cmp     al,0ffh     } する
0113  74 1D                            jz      nofile      } 該当ファイルがなければ "nofile"
                                                           } へジャンプする

0115  BB 0080 R               nxtfile:mov     bx,offset dta
0118  B9 0025                         mov     cx,37       } DTAに返されたディレクトリ情報
011B  E8 0000 E                        call    dump        } (37バイト)をダンプ形式で出力する
011E  E8 0000 E                        call    crlfout     }

0121  B4 12                            mov     ah,12h      } "次のディレクトリ・エントリの検
0123  BA 005C R                        mov     dx,offset fcb } 索" のシステムコール
0126  CD 21                            int     21h         }
0128  3C FF                            cmp     al,0ffh     } 該当ファイルが存在すれば
012A  75 E9                            jne     nxtfile     } "nxtfile" へジャンプする

012C  BA 0147 R                        mov     dx,offset msged } 1つでも該当するファイルが
012F  EB 04 90                         jmp     msgout      } あれば "End File" と出力
                                                           } する

0132  BA 013B R               nofile: mov     dx,offset msgnof } "No File" のメッセージ

0135  B4 09                   msgout: mov     ah,9        } メッセージ出力
0137  CD 21                           int     21h         }

0139  CD 20                           int     20h ..... プログラムの終了
                              ;
013B  0D 0A 4E 6F 20 46       msgnof  db  0dh,0ah,'No File',0dh,0ah,'$'
      69 6C 65 0D 0A 24
0147  0D 0A 45 6E 64 20       msged   db  0dh,0ah,'End File',0dh,0ah,'$'
      46 69 6C 65 0D 0A
      24
                              ;
0154                          cseg    ends
                              ;
                                      end     start
```

Fig. 7.15　Sample List　0DH, 0EH, 11H, 12H

実行手順　　FUNC−0D [〈ドライブ名：〉]〈ファイル名〉 ↵

```
A>dir↵

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM     17190  83-09-19  16:19
BIN          <DIR>      84-02-22  20:32
COM          <DIR>      84-02-22  20:21
AUTOEXEC BAT        61  84-02-25  17:22
CONFIG   SYS        70  84-03-20  21:07
FUNC         <DIR>      84-02-22  22:04
FACT     EXE     13496  84-03-05  19:14
TEST              160  84-02-29  21:21
SUBDIR1      <DIR>      84-02-23   1:19
CAL          <DIR>      84-02-23   1:40
TXT          <DIR>      84-02-23   5:07
ASMWORK      <DIR>      84-02-25   7:22
       12 個のファイルがあります
   715776 バイトが使用可能です
```

カレント・ディレクトリを表示して確認する
DIRコマンドでは表示されないが実際は,
"IO.SYS","MSDOS.SYS"が存在する

```
A>b:↵ ……カレント・ドライブをドライブBにする

B>func-0d *.*↵ ……"FUNC-0D.COM"を起動しすべてのファイルを検索する
01 43 4F 4D 4D 41 4E 44 20 43 4F 4D 20 00 00 00   .COMMAND COM ...
00 00 00 00 00 00 00 6E 82 33 07 3C 00 26 43 00   .......n.3.<.&C.
00 61 6D 2E 0D 00 00 00 00 00 00 00 00 00 00 00   .am.............
└ DTA内にセットされるデータ
01 41 55 54 4F 45 58 45 43 42 41 54 20 00 00 00   .AUTOEXECBAT ...
00 00 00 00 00 00 00 D9 8A 59 08 9E 01 3D 00 00   .......ﾙ.Y...=..
00 61 6D 2E 0D 00 00 00 00 00 00 00 00 00 00 00   .am.............
┌ ドライブコード(Aドライブ) ┌ ファイル名    ┌ 拡張子  ┌ 属性
01 43 4F 4E 46 49 47 20 20 53 59 53 20 00 00 00   .CONFIG  SYS ...
00 00 00 00 00 00 00 E5 A8 74 08 C2 01 46 00 00   .......ｨt.ﾂ.F..
00 61 6D 2E 0D 00 00 00 00 00 00 00 00 00 00 00   .am.............
                        ファイルの時刻 ファイルの日付  ファイルサイズ
01 46 41 43 54 20 20 20 20 45 58 45 20 00 00 00   .FACT    EXE ...
00 00 00 00 00 00 00 C1 99 65 08 9F 01 B8 34 00   .......ﾁ.e...ｸ4.
00 61 6D 2E 0D 00 00 00 00 00 00 00 00 00 00 00   .am.............

01 54 45 53 54 20 20 20 20 20 20 20 20 00 00 00   .TEST           ...
00 00 00 00 00 00 00 BD AA 5D 08 03 02 A0 00 00   .......ｽｪ]... ..
00 61 6D 2E 0D 00 00 00 00 00 00 00 00 00 00 00   .am.............

End File
A>■……ドライブAにカレント・ドライブが変更されている
```

FCBによる検索なので通常のファイルのみしか見つけられない

Fig. 7.16 実行結果 0DH, 0EH, 11H, 12H

ここで拡張FCBの使用例を示すため，上記のFUNC−0Dによるディレクトリ・エントリの検索を拡張FCBによるものに置き換えて，次のサンプル・プログラムFUNC−11を作成します．

FUNCTION 11H, 12H Sample Program

ディレクトリ・エントリの検索に直接必要ない部分（ファンクション0DHおよび0EHを使用している部分）以外は，FUNC−0Dと同様に指定されたファイルを検索し，ファイルに関する情報を表示する．ただし，拡張FCBの属性には1FHをセットしてあるので，FUNC−0Dでは検索されない隠されたファイルやディレクトリおよびボリュームラベルも検索され，拡張FCBのフォーマットで情報が表示される．

```
                        ;-----------------------------------------;
                        ;   Function  11h: Search For First Entry ;
                        ;             12h: Search For Next Entry  ;
                        ;-----------------------------------------;
                        ;
0000                    cseg    segment
                                extrn   dump:near,crlfout:near
                                assume  cs:cseg,ds:cseg
                        ;
005C                            org     5ch
                        ;
005C    24 [            fcb     db      36 dup (?)
```

"dump,"crlfout"を外部サブルーチンとして定義する

```
                ??
                    ]

                                    ;
0080        80 [                dta     db      128 dup (?)
                ??
                    ]

                                    ;
0100                                        org     100h
                                    ;                                   デフォルトFCBにセットされたドライブ名およびファイル名を拡張FCBの先頭アドレス+07Hからの12バイトに転送する
0100  BE 005C R                     start:  mov     si,offset fcb
0103  BF 0169 R                             mov     di,offset extfcb[7]
0106  B9 000C                              mov     cx,12
0109  FC                                   cld
010A  F3/ A4                           rep  movsb
010C  C6 06 0162 R FF 90                    mov     extfcb[0],0ffh      ……拡張FCBの先頭はFFH
0112  C6 06 0168 R 1F 90                    mov     extfcb[6],1fh       ……すべてのディレクトリ・エントリを検索する

0118  B4 11                                 mov     ah,11h              "最初のディレクトリ・エントリの検索"のシステムコール
011A  BA 0162 R                             mov     dx,offset extfcb
011D  CD 21                                 int     21h
011F  3C FF                                 cmp     al,0ffh             該当ファイルがなければ"nofile"へジャンプする
0121  74 1D                                 jz      nofile

0123  BB 0080 R                     nxtfile:mov     bx,offset dta       DTAに返されたディレクトリ情報(44バイト)をダンプ形式で出力する
0126  B9 002C                               mov     cx,44
0129  E8 0000 E                            call    dump
012C  E8 0000 E                            call    crlfout

012F  B4 12                                 mov     ah,12h              "次のディレクトリ・エントリの検索"のシステムコール
0131  BA 0162 R                             mov     dx,offset extfcb
0134  CD 21                                 int     21h
0136  3C FF                                 cmp     al,0ffh             該当するファイルがあれば"nxtfile"へジャンプする
0138  75 E9                                 jne     nxtfile

013A  BA 0155 R                             mov     dx,offset msged     1つでも該当するファイルがあれば"End File"と出力する
013D  EB 04 90                              jmp     msgout

0140  BA 0149 R                     nofile: mov     dx,offset msgnof    "No File"のメッセージ

0143  B4 09                         msgout: mov     ah,9                メッセージ出力
0145  CD 21                                 int     21h

0147  CD 20                                 int     20h ……… プログラムの終了
                                    ;
0149  0D 0A 4E 6F 20 46             msgnof  db  0dh,0ah,'No File',0dh,0ah,'$'
      69 6C 65 0D 0A 24
0155  0D 0A 45 6E 64 20             msged   db  0dh,0ah,'End File',0dh,0ah,'$'
      46 69 6C 65 0D 0A
      24
0162      2C [                      extfcb  db      44 dup (?)
                ??
                    ]

                                    ;
018E                                cseg    ends
                                    ;
                                            end     start
```

Fig. 7.17　Sample List　11H, 12H

実行手順　　FUNC－11 [〈ドライブ名：〉]〈ファイル名〉 ↵

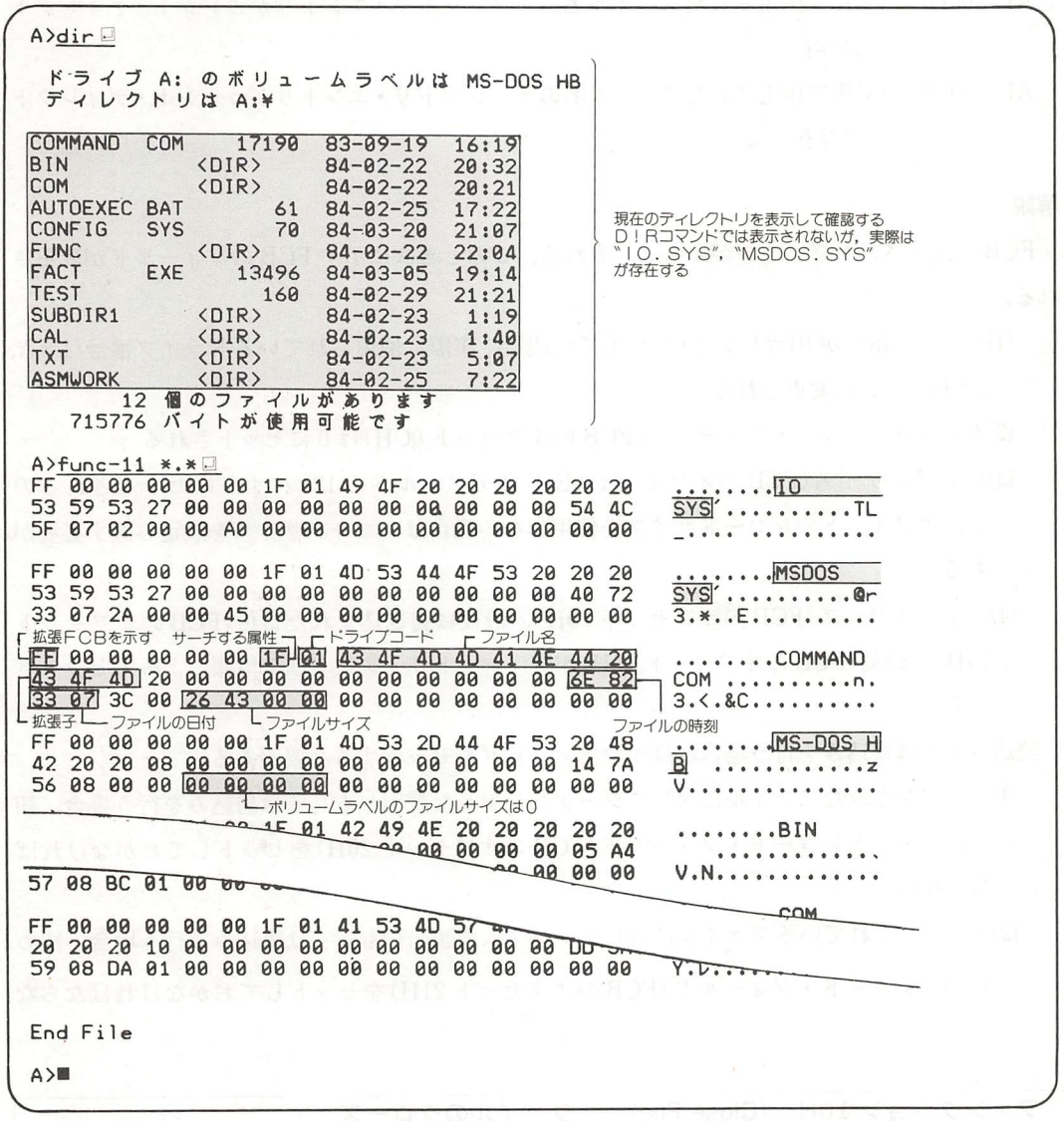


Fig. 7.18　実行結果　11H, 12H

FUNCTION　0FH, 10H, 14H, 15H, 16H

ファンクション 0FH　(Open File)——ファイルのオープン

コール手順

AH　　←0FH

DS：DX　←オープンされていない FCB の先頭アドレス

INT　21H

Return

AL＝00H　FCBで指定されたファイル名のディレクトリ・エントリがカレント・ディレクトリに存在した

AL＝FFH　FCBで指定されたファイル名のディレクトリ・エントリがカレント・ディレクトリに存在しない

解説

FCBで指定されたファイルがオープンされる．このとき次に示すFCBのフィールドが変更される．

(1)ドライブ番号が0(カレント・ドライブ)の場合，実際に使用されているドライブ番号(1＝A, 2＝B, ……)に変更される

(2)カレントブロック・フィールド(FCBのオフセット0CH)は0にセットされる

(3)レコードサイズ(FCBのオフセット0EH)はデフォルトの128バイトにセットされるので，デフォルトのレコードサイズを使用しない場合はレコードサイズを設定し直す必要がある

(4)ファイルサイズ(FCBのオフセット10H)，最後に書き込まれた日付(FCBのオフセット14H)と時刻(FCBのオフセット16H)がディレクトリ・エントリの情報によってセットされる

読込みまたは書込みを行う場合には次のフィールドをセットする必要がある．

(1)オープンされたファイルに対してシーケンシャルな読出しもしくは書込みを行う場合，初めにカレントレコード・フィールド(FCBのオフセット20H)をセットしておかなければならない

(2)オープンされているファイルに対してランダムな読出しもしくは書込みを行う場合，初めに相対レコード・フィールド(FCBのオフセット21H)をセットしておかなければならない

ファンクション10H　(Close File)——ファイルのクローズ

コール手順

```
AH      ←10H
DS：DX  ←オープンされているFCBの先頭アドレス
INT  21H
```

Return

AL＝00H　ディレクトリ・エントリが存在する

AL＝FFH　ディレクトリ・エントリが存在しない

解説

ファイルのオープン(ファンクション 0FH)あるいはファイルの作成(ファンクション 16H)を使用して，FCB で示されるファイルに書込みが行われた場合，ディレクトリ・エントリを更新するためにはこのシステムコールを行わなければならない．ディレクトリ・エントリが存在する場合，FCB のディレクトリ・エントリと対応するファイル情報を比較して，違いがあれば FCB と一致するようにディレクトリ・エントリを更新する．

ファンクション 14H　(Sequential Read)——シーケンシャルな読出し

コール手順

```
AH      ←14H
DS：DX  ←オープンされている FCB の先頭アドレス
INT  21H
```

Return

AL＝00H　読出しは正常に行われた

AL＝01H　ファイルの終わり(EOF)を検出した．このレコードにデータははいっていない

AL＝02H　DTA 内に 1 レコードを読み込むのに十分な領域がなく，読出しは取り消された

AL＝03H　読出しの途中でファイルの終わり(EOF)を検出した．EOF までのデータが DTA 内に読み込まれ，レコードの残りの部分が 00H で埋められた

解説

ファイルから FCB で示されるレコードサイズに等しいバイト数のデータが読み込まれ，DTA に格納される．読み出されるレコードのポインタは，1 ワードのカレントブロック(FCB のオフセット 0CH)と 1 バイトのカレントレコード(FCB のオフセット 20H)の各フィールドの値によって示される．レコードの読込みが終了すると，カレントレコード・フィールドが次のレコードを示すように 1 だけインクリメントされる．

ファンクション 15H　(Sequential Write)——シーケンシャルな書込み

コール手順

```
AH      ←15H
DS：DX  ←オープンされている FCB の先頭アドレス
INT  21H
```

Return

AL＝00H　書込みは正常に行われた

AL＝01H　ディスクに空き領域がなく，書込みが中止された

AL＝02H　DTA にレコードを書き込むための十分な領域がなく，書込みは中止された

解説

DTA から FCB のオフセット 0EH の 1 ワードで示されるレコードサイズに等しいバイト数のデータがファイルに書き込まれる．このときカレントブロック(FCB のオフセット 0CH)とカレントレコード(FCB のオフセット 20H)の各フィールドの値によって示されるレコードにデータが書き込まれる．カレントレコード・フィールドは次のレコードを示すように 1 だけインクリメントされ，次の書込みに備える．

ファンクション 16H　(Create File)——ファイルの作成

コール手順

```
AH        ←16H
DS：DX    ←オープンされていない FCB の先頭アドレス
INT  21H
```

Return

AL＝00H　ファイルが作成(ディレクトリ・エントリがディスクに登録)された

AL＝FFH　ディレクトリに空き領域がなく，指定されたファイル名のディレクトリ・エントリも存在しない

解説

DS：DX の指すオープンされていない FCB で指定したファイルが作成(ディレクトリ・エントリをディスクに登録)され，オープン(ファンクション 0FH)される．拡張 FCB を使用して，アトリビュートバイト(FCB のオフセット－1)のビット 1 をセットすれば，隠されたファイルを作成できる．FCB で指定されたファイル名のディレクトリ・エントリが存在した場合，既存のファイルは削除され，大きさ 0 のファイルが作成される．指定されたファイル名のディレクトリ・エントリが存在しない場合，ディレクトリに空き領域があればファイルを作成してファイルの大きさを 0 に初期設定する．

FUNCTION　0FH, 10H, 14H, 15H, 16H　Sample Program

COPY コマンドのカレント・ディレクトリ内でのコピーの機能の一部に相当するプログラムを作成する．レコードサイズは 128 バイトとし，1 レコードごとに読出し／書込みを行う．

```
                                ;-----------------------------------------------;
                                ;   Function  0fh: Open File                   ;
                                ;             10h: Close File                   ;
                                ;             14h: Sequential Read              ;
                                ;             15h: Sequential Write             ;
                                ;             16h: Create File                  ;
                                ;-----------------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
005C                                    org     5ch
                                ;
005C    25 [                    fcb_s   db      37 dup (?)
            ??
                ]

                                ;
0100                                    org     100h
                                ;
0100  B9 0010                   start:  mov     cx,10h
0103  BE 006C R                         mov     si,offset fcb_s[16]
0106  BF 01D2 R                         mov     di,offset fcb_d
0109  1E                               push    ds
010A  07                                pop     es
010B  FC                                cld
010C  F3/ A4                      rep   movsb

010E  B4 0F                             mov     ah,0fh
0110  BA 005C R                         mov     dx,offset fcb_s
0113  CD 21                             int     21h
0115  3C FF                             cmp     al,0ffh
0117  74 2F                             je      openerr

0119  B4 16                             mov     ah,16h
011B  BA 01D2 R                         mov     dx,offset fcb_d
011E  CD 21                             int     21h
0120  3C FF                             cmp     al,0ffh
0122  74 2A                             je      makeerr

0124  32 C0                             xor     al,al
0126  A2 007C R                         mov     fcb_s[32],al
0129  A2 01F2 R                         mov     fcb_d[32],al

012C  B4 14                     nxtrec: mov     ah,14h
012E  BA 005C R                         mov     dx,offset fcb_s
0131  CD 21                             int     21h
0133  3C 03                             cmp     al,3
0135  74 04                             je      lastrec
0137  3C 00                             cmp     al,0
0139  75 1F                             jne     cpyend

013B  B4 15                     lastrec:mov     ah,15h
013D  BA 01D2 R                         mov     dx,offset fcb_d
0140  CD 21                             int     21h
0142  3C 00                             cmp     al,0
0144  75 0E                             jne     wrterr
0146  EB E4                             jmp     nxtrec

0148  BA 0174 R                openerr:mov     dx,offset msgser
014B  EB 21 90                          jmp     msgout

014E  BA 018E R                 makeerr:mov     dx,offset msgmer
0151  EB 1B 90                         jmp     msgout

0154  BA 01A1 R                 wrterr: mov     dx,offset msgwer
0157  EB 15 90                         jmp     msgout
```

デフォルトFCB内にセットされるディスティネーション側のファイル名(FCB内のオフセット6CHから格納されている)を"fcb_d"に転送する

ソース側のFCB("fcb_s")で示されるファイルをオープンする
ファイルが存在しない場合は,"openerr"へジャンプする

ディスティネーション側のFCB("fcb_d")で示されるファイルを新たに作成するとともにオープンする. ディレクトリに空き領域がなく, ファイルが作成できない場合は"makeerr"へジャンプする

ソース側およびディスティネーション側のカレントレコード・フィールドを00Hにセットし, 最初のレコードから読出し/書込みを行う

ソースファイルを1レコード(デフォルトのレコードサイズ80H/バイト)読み込む. データはデフォルトDTAのPSP:80Hに読み込まれる. ファイルの最後のレコードにデータが途中まではいっている場合は"lastrec"へジャンプする. ファイルの終わりの場合は"cpyend"へジャンプする

DTAの1レコード分のデータをディスティネーションファイルに書き込む. ディスクがいっぱいになった場合は,"Wrterr"へジャンプする

"nxtrec"へ戻って次のレコードの読込みを行う

ソース・ファイルがない場合のエラーメッセージ

ディスティネーションファイルが作成できない場合のエラーメッセージ

ディスクフルになった場合のエラーメッセージ

```
015A  B4 10                     cpyend: mov     ah,10h
015C  BA 01D2 R                         mov     dx,offset fcb_d
015F  CD 21                             int     21h
0161  3C FF                            cmp     al,0ffh
0163  74 06                            je      clserr
0165  BA 01C0 R                         mov     dx,offset msgend
0168  EB 04 90                          jmp     msgout

016B  BA 01AF R                 clserr: mov     dx,offset msgcer

016E  B4 09                     msgout: mov     ah,9
0170  CD 21                             int     21h

0172  CD 20                             int     20h
                                ;
0174  0D 0A 53 6F 75 72         msgser  db  0dh,0ah,'Source File Not Found',0dh
                                ,0ah,'$'
      63 65 20 46 69 6C
      65 20 4E 6F 74 20
      46 6F 75 6E 64 0D
      0A 24
018E  0D 0A 44 69 72 65         msgmer  db  0dh,0ah,'Directory Full',0dh,0ah,'$
      63 74 6F 72 79 20
      46 75 6C 6C 0D 0A
      24
01A1  0D 0A 44 69 73 6B         msgwer  db  0dh,0ah,'Disk Full',0dh,0ah,'$'
      20 46 75 6C 6C 0D
      0A 24
01AF  0D 0A 43 61 6E 6E         msgcer  db  0dh,0ah,'Cannot Close',0dh,0ah,'$'
      6F 74 20 43 6C 6F
      73 65 0D 0A 24
01C0  0D 0A 43 6F 70 79         msgend  db  0dh,0ah,'Copy Complete',0dh,0ah,'$'
      20 43 6F 6D 70 6C
      65 74 65 0D 0A 24
                                ;
01D2      25 [                  fcb_d   db      37 dup (?)
            ??
               ]

                                ;
01F7                            cseg    ends
                                ;
                                        end     start
```

コピーが終了したのでディスティネーションファイルをクローズする．クローズできない場合には"clserr"へジャンプする

コピー終了のメッセージ

クローズ・エラーのメッセージ

メッセージの出力

プログラムの終了

Fig. 7.19　Sample List　0FH，10H，14H，15H，16H

実行手順　　FUNC−0F　[〈ドライブ名：〉]〈ソースファイル名〉　[〈ドライブ名：〉]〈ディスティネーションファイル名〉 ↵

```
A>dir func-0f* ↵ ……コピーするソースファイルを確認する

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

FUNC-0F  ASM      1517  84-03-24  18:01
FUNC-0F  COM       247  84-03-06  20:10
       2 個のファイルがあります
  253952 バイトが使用可能です

A>func-0f func-0f.asm func-0fh.asm ↵ ……"FUNC-0F．ASM"を"FUNC-0FH．ASM"にコピーする

Copy Complete ……コピーが完了したメッセージ

A>dir func-0f* ↵ ……新しくファイルが作られたことの確認
```

```
 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

FUNC-0F  ASM      1517  84-03-24  18:01
FUNC-0F  COM       247  84-03-06  20:10
FUNC-0FH ASM      1536  84-03-24  18:02
       3 個のファイルがあります
  251904 バイトが使用可能です

A>fc /b func-0f.asm func-0fh.asm
F2 にまだデータが残っています
A>■
```

ファイルの大きさが異なることに注意
ディスティネーションには128バイト単位で書き込むため最後に余分なデータが付いてしまう

FCでファイルを比較する

"FUNC-0FH. ASM" に余分なデータが付いてしまうためこのメッセージが出るが，ほかは正しくコピーされている

Fig. 7.20 実行結果 0FH, 10H, 14H, 15H, 16H

ディスティネーションファイルに既存のファイルを指定すると，その内容がソースファイルの内容に置き換えられることを確認してください．

FUNCTION 13H

ファンクション13H (Delete File)——ファイルの削除

コール手順

AH ←13H
DS：DX ←オープンされていないFCBの先頭アドレス
INT 21H

Return

AL＝00H FCBで指定されたファイル名のディレクトリ・エントリが存在し，そのファイルが削除された

AL＝FFH ディレクトリ・エントリが存在しない．あるいはファイルに削除できない属性が与えられていた

解説

FCBで指定されたファイル名の，すべてのディレクトリ・エントリが削除される．このとき，ファイル名にはワイルドカードが使用できるので，一致するすべてのファイルを削除することができる．

FUNCTION 13H Sample Program

DELコマンドの機能の一部を実現するプログラムを作成する．PSPのオフセット5CHにコマンドラインのファイル名がセットされるのを利用して，そのファイルを削除する．

```
                                ;-------------------------------------;
                                ;   Function  13h: Delete File        ;
                                ;-------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
005C                                    org     5ch
                                ;
005C    25 [                    fcb     db      37 dup (?)
            ??
                ]

                                ;
0100                                    org     100h
                                ;
0100  B4 13                     start:  mov     ah,13h          ┐"ファイルの削除" のシステムコー
0102  BA 005C R                         mov     dx,offset fcb   │ル．FCBはデフォルトのPSP：
0105  CD 21                             int     21h             ┘ 5CHとし，コマンドラインで与え
                                                                  えたファイルを削除する
0107  BA 0117 R                         mov     dx,offset msgok ┐
010A  3C 00                             cmp     al,0            │
010C  74 03                             je      msgout          │指定されたファイルが削除
010E  BA 011E R                         mov     dx,offset msgerr│された場合は "Ok" と表
0111  B4 09                     msgout: mov     ah,9            │示し，指定したファイルが
0113  CD 21                             int     21h             │存在しないか，削除できな
                                                                  │い属性(リードオンリー，デ
0115  CD 20                             int     20h ……プログラムの終了 │ィレクトリ) を持っていた
                                ;                               │場合は "Error" と表示す
0117  0D 0A 4F 6B 0D 0A        msgok   db  0dh,0ah,'Ok',0dh,0ah,'$'  │る
      24                                                        ┘
011E  0D 0A 45 72 72 6F        msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
                                ;
0128                            cseg    ends
                                ;
                                        end     start
```

Fig. 7.21　Sample List　13H

実行手順　　FUNC−13　[〈ドライブ名：〉]〈ファイル名〉 ↵

```
A>dir func-13↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

FUNC-13  ASM        484  84-03-06  17:59   ┐
FUNC-13  COM         40  84-03-06  18:00   │"FUNC-13.*" に該当するファイルを確認しておく
FUNC-13  LST       1671  84-03-06  18:00   ┘
       3 個のファイルがあります
  280576 バイトが使用可能です

A>func-13 func-13.??m↵ ……"FUNC-13.COM" の起動．"FUNC-13.ASM"と"FUNC-13.COM"のみを削除する

Ok …… 正常終了したメッセージ

A>dir func-13↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

FUNC-13  LST       1671  84-03-06  18:00   ┐"FUNC-13.LST" のみが残っている
       1 個のファイルがあります
  282624 バイトが使用可能です

A>■
```

Fig. 7.22　実行結果　13H

このプログラムは，ライトプロテクトがかけてあるファイルは削除できませんが，拡張 FCB を使用すればそのようなファイルでも削除できます．

FUNCTION 17H

ファンクション 17H (Rename File)——ファイル名の変更

コール手順

```
AH      ← 17H
DS：DX  ← FCB の先頭アドレス
INT  21H
```

Return

AL＝00H　ディレクトリ・エントリが存在する

AL＝FFH　ディレクトリ・エントリが存在しないか，あるいは新しいファイル名がすでに存在している

解説

このシステムコールは，ファイルの名前を付け換えるのに使用される．

DS：DX の示す FCB は，Fig. 7.23 のようなフォーマットになっていなければならない．

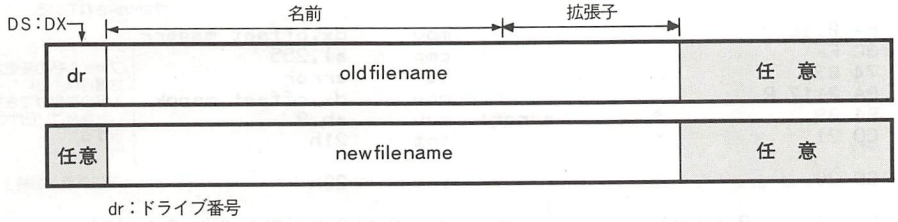


Fig. 7 .23　ファイル名の付換え

このとき，oldfilename を newfilename に付け換える．

oldfilename のドライブ番号は，00H がカレント・ドライブ，01H が A，02H が B……を表しており，このとき newfilename のドライブ番号は無視される．なおファイル名にはワイルドカードが使用でき，該当するファイルは，すべて名前が付け換えられる．newfilename 内のワイルドカードの部分には，名前を付け換えようとするファイルと同じもの(oldfilename)が採用される．このとき使用できるワイルドカードは〝?〟のみである．〝*〟を使うと，ファイル名に〝*〟の付いたファイルが作られてしまう．

例：oldfilename　T E S T␣␣␣␣A S M

　　newfilename　？？？？？N E W？？？

このとき実際に付けられるファイル名は以下のようになる．

```
TEST␣NEWASM
```

FUNCTION　17H　　Sample Program

REN コマンドの機能の一部を実現するプログラムを作成する．このプログラムは，カレント・ディレクトリ内だけで有効である．PSP のデフォルト FCB 内のオフセット 5CH および 6CH に，コマンドラインのファイル名がセットされるのを利用している．

```
                                ;----------------------------------------;
                                ;   Function  17: Rename File            ;
                                ;----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
005C                                    org     5ch
                                ;
005C    25 [                    fcb     db      37 dup (?)
            ??
               ]

                                ;
0100                                    org     100h
                                ;
0100  B4 17                     start:  mov     ah,17h          ; "ファイル名の変更" のシステムコール.FCBはデフォルトのPSP：
0102  BA 005C R                         mov     dx,offset fcb   ; 5CHとする．FCBの5CHと6
0105  CD 21                             int     21h             ; CHにコマンドラインのファイル名
                                                                ; がセットされている
0107  BA 011E R                         mov     dx,offset msgerr
010A  3C FF                             cmp     al,255          ; ファイル名の変更が終了し
010C  74 03                             je      error           ; た場合は "Ok"，ファイ
010E  BA 0117 R                         mov     dx,offset msgok ; ル名の変更ができずに終了
0111  B4 09                     error:  mov     ah,9            ; した場合は "Error" と表
0113  CD 21                             int     21h             ; 示する

0115  CD 20                             int     20h             ; プログラムの終了
                                ;
0117  0D 0A 4F 6B 0D 0A         msgok   db  0dh,0ah,'Ok',0dh,0ah,'$'
      24
011E  0D 0A 45 72 72 6F         msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
                                ;
0128                            cseg    ends
                                ;
                                        end     start
```

Fig. 7.24　Sample List　17H

実行手順　　FUNC−17 [〈ドライブ名：〉]〈oldfilename〉〈newfilename〉 ↵

newfilename のドライブ名は付けても無視される

Return

BX ←使用可能なクラクタ数

DX ← 1 ドライブあたりのクラスタ数

CX ← 1 セクタあたりのバイト数

AX ← 1 クラスタあたりのセクタ数

AX ← FFFFH(ドライブ番号が無効である場合)

解説

DL レジスタで指定したドライブに対するディスクの情報を返す. MS-DOS は, FAT をクラスタ単位で管理しているが, 1 クラスタが何セクタに相当するかがこのシステムコールによって得られる. また次の式で示されるバイト数が, ディスクの使用可能なバイト数となる.

(使用可能なクラスタ数) * (1 クラスタあたりのセクタ数) * (1 セクタあたりのバイト数)

ファンクション 38H (Return Country-Dependent Information) ―― 国別情報の読出し

コール手順

AH ← 38H

DS : DX ←返される 32 バイトの情報に対するバッファの先頭アドレス

DX ← FFFFH(現在の国を AL レジスタで示す国にセットする場合)

AL ←情報を得ようとする国のカントリーコード
(00H…現在の国, 01H…USA, 51H…日本)
DX が FFFFH の場合 : 現在の国とするカントリーコード

INT 21H

Return

キャリーがセットされた場合

AX = 02H 指定された国のためのテーブルが存在しない

キャリーがセットされない場合

DX = FFFFH AL レジスタで指定した国が現在の国となる

それ以外 DS : DX で指定されたアドレスから 32 バイトのメモリ領域に, 国際適用業務に該当する情報が返される

解説

DS : DX で指定されたアドレスから 32 バイトのメモリ領域に, 国際適用業務に該当する Fig. 7.26 のフォーマットの情報が返される. AL レジスタに与えるカントリーコードは通常国際電話プリフィックスコードを用いる.

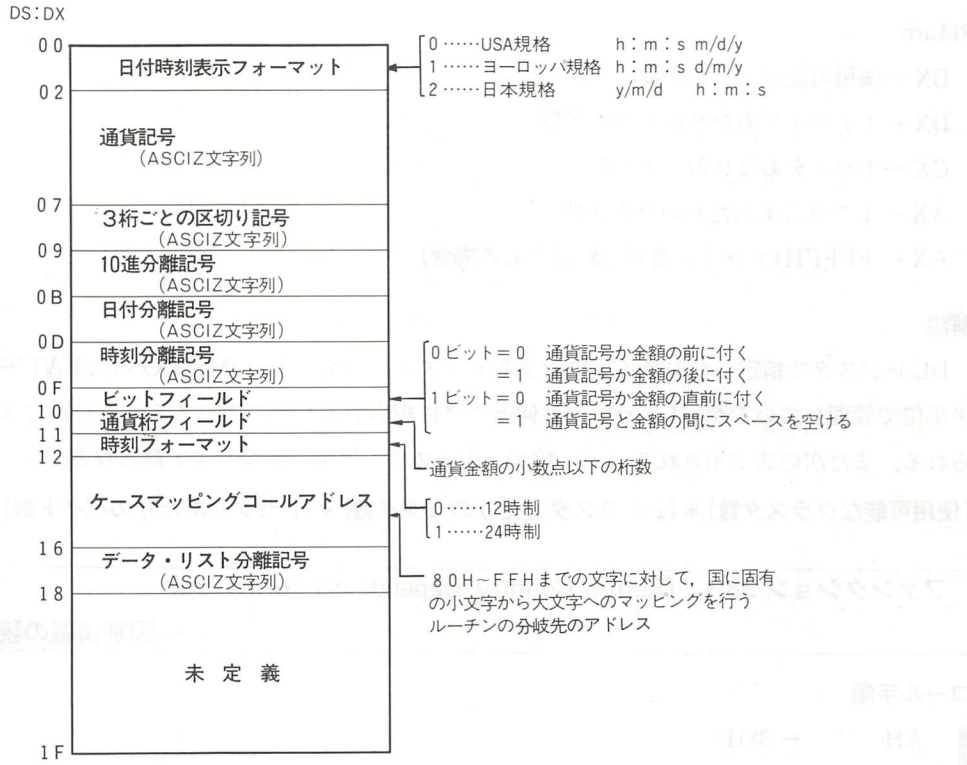


Fig. 7 .26　国別情報の取出し

DX レジスタに FFFFH をセットしてコールすると AL レジスタで指定した国が現在の国となり，DIR コマンドで表示される日時のフォーマットも指定した国のものとなる．指定した国のデータが存在しない場合には，現在の国は変更されない．

FUNCTION　19H, 30H, 36H, 38H　Sample Program

これらのファンクションを用いて得られる情報を表示するプログラムを作成する．まず，ファンクション 30H を用いて MS-DOS のバージョン番号，19H を用いてカレント・ドライブ，36H を用いてカレント・ドライブの情報，38H を用いて日本の国別情報を表示する．

```
                        ;-----------------------------------------------;
                        ;   Function  19h: Current Disk                 ;
                        ;             30h: Get DOS Version Number       ;
                        ;             36h: Get Disk Free Space          ;
                        ;             38h: Return Country_Dependent     ;
                        ;                                Information    ;
                        ;-----------------------------------------------;
                        ;
0000                    cseg    segment
                                extrn   putch:near,hexout:near
                                extrn   dump:near
                                assume  cs:cseg,ds:cseg
```

"putch","hexout","dump"を外部サブルーチンとして定義する

```
                                ;
0100                            org     100h
                                ;
0100  B4 09                     start:  mov     ah,9
0102  BA 0187 R                         mov     dx,offset msgver
0105  CD 21                             int     21h
0107  B4 30                             mov     ah,30h
0109  CD 21                             int     21h
010B  0C 30                             or      al,30h
010D  E8 0000 E                         call    putch
0110  B0 2E                             mov     al,'.'
0112  E8 0000 E                         call    putch
0115  8A C4                             mov     al,ah
0117  D4 0A                             aam
0119  0D 3030                           or      ax,3030h
011C  86 C4                             xchg    al,ah
011E  E8 0000 E                         call    putch
0121  8A C4                             mov     al,ah
0123  E8 0000 E                         call    putch

0126  B4 09                             mov     ah,9
0128  BA 0199 R                         mov     dx,offset msgcdsk
012B  CD 21                             int     21h
012D  B4 19                             mov     ah,19h
012F  CD 21                             int     21h
0131  04 41                             add     al,41h
0133  E8 0000 E                         call    putch

0136  B4 36                             mov     ah,36h
0138  B2 00                             mov     dl,0
013A  CD 21                             int     21h
013C  52                                push    dx
013D  50                                push    ax
013E  B4 09                             mov     ah,9
0140  BA 01AD R                         mov     dx,offset dskf1
0143  CD 21                             int     21h
0145  58                                pop     ax
0146  E8 0000 E                         call    hexout
0149  B4 09                             mov     ah,9
014B  BA 01CE R                         mov     dx,offset dskf2
014E  CD 21                             int     21h
0150  8B C1                             mov     ax,cx
0152  E8 0000 E                         call    hexout
0155  B4 09                             mov     ah,9
0157  BA 01E5 R                         mov     dx,offset dskf3
015A  CD 21                             int     21h
015C  8B C3                             mov     ax,bx
015E  E8 0000 E                         call    hexout
0161  B4 09                             mov     ah,9
0163  BA 0214 R                         mov     dx,offset dskf4
0166  CD 21                             int     21h
0168  58                                pop     ax
0169  E8 0000 E                         call    hexout

016C  B4 09                             mov     ah,9
016E  BA 0244 R                         mov     dx,offset msginf
0171  CD 21                             int     21h
0173  B4 38                             mov     ah,38h
0175  BA 0257 R                         mov     dx,offset block
0178  B0 51                             mov     al,51h
017A  CD 21                             int     21h
017C  BB 0257 R                         mov     bx,offset block
017F  B9 0020                           mov     cx,32
0182  E8 0000 E                         call    dump

0185  CD 20                             int     20h
```

MS-DOSのバージョンを□.□□と表示する

カレント・ドライブは"A","B"…で表示する

"ディスクの残り容量の読出し"のシステムコール

上記のシステムコールで返されるAX, BX, CX, DXレジスタの値をメッセージとともに16進4桁で表示する

国別情報をダンプ形式で出力する
ALレジスタに51Hをセットして日本の情報を得て"block"から32バイトをダンプ形式で表示する

プログラムを終了する

```
                                        ;
0187  0D 0A 4D 53 2D 44                 msgver  db  0dh,0ah,'MS-DOS Version $'
      4F 53 20 56 65 72
      73 69 6F 6E 20 24
0199  0D 0A 43 75 72 72                 msgcdsk db  0dh,0ah,'Current Drive is $'
      65 6E 74 20 44 72
      69 76 65 20 69 73
      20 24
01AD  0D 0A 53 65 63 74                 dskf1   db  0dh,0ah,'Sector per allocation unit
                                         .. $'
      6F 72 20 70 65 72
      20 61 6C 6C 6F 63
      61 74 69 6F 6E 20
      75 6E 69 74 20 2E
      2E 20 24
01CE  0D 0A 42 79 74 65                 dskf2   db  0dh,0ah,'Bytes per sector .. $'
      73 20 70 65 72 20
      73 65 63 74 6F 72
      20 2E 2E 20 24
01E5  0D 0A 4E 75 6D 62                 dskf3   db  0dh,0ah,'Number of free allocation
                                        units on drive .. $'
      65 72 20 6F 66 20
      66 72 65 65 20 61
      6C 6C 6F 63 61 74
      69 6F 6E 20 75 6E
      69 74 73 20 6F 6E
      20 64 72 69 76 65
      20 2E 2E 20 24
0214  0D 0A 54 6F 74 61                 dskf4   db  0dh,0ah,'Total number of allocation
                                         units on drive .. $'
      6C 20 6E 75 6D 62
      65 72 20 6F 66 20
      61 6C 6C 6F 63 61
      74 69 6F 6E 20 75
      6E 69 74 73 20 6F
      6E 20 64 72 69 76
      65 20 2E 2E 20 24
0244  0D 0A 4A 61 70 61                 msginf  db  0dh,0ah,'Japan Standard',0dh,0ah,'$
                                        '
      6E 20 53 74 61 6E
      64 61 72 64 0D 0A
      24
                                        ;
0257     20 [                           block   db      32 dup (?)
            ??
                ]

                                        ;
0277                                    cseg    ends
                                        ;
                                                end     start
```

Fig. 7.27　Sample List　19H, 30H, 36H, 38H

実行手順　　FUNC−19 ↵

```
A>dir func-19.* ↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

FUNC-19  ASM      1664  84-02-29  20:20
FUNC-19  COM       534  84-02-29  20:29
        2 個のファイルがあります
   283648 バイトが使用可能です
```

ディスクの空き領域の大きさを確認しておく

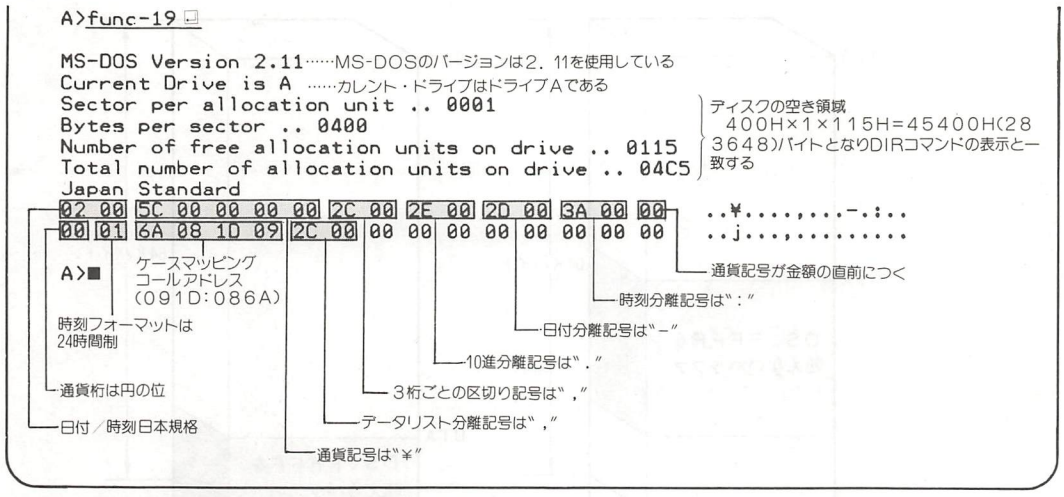


Fig. 7.28 実行結果 19H, 30H, 36H, 38H

FUNCTION 1AH, 21H, 22H, 23H

ファンクション1AH (Set Disk Transfer Address)

——ディスク転送アドレスの設定

コール手順

```
AH      ←1AH
DS：DX  ←DTA
INT  21H
```

Return

なし

解説

ディスクからの読出し／書込み(14H, 15H, 21H, 22H, 27H, 28Hの各ファンクション)におけるバッファの位置が，このシステムコールにより指定される．また，ディレクトリの検索(11H, 12H, 4EH, 4FHの各ファンクション)を行う場合に返されるディレクトリ情報の格納先のアドレスをセットすることができる．

バッファに十分な領域を取ってあっても，DTAの2つのセグメントにわたって，ディスクからの読出し／書込みを行うことはできない．すなわち，このシステムコールで，読出し／書込みを行うとDS：FFFFHの範囲を超えるようなアドレスにDTAをセットしても，実際に読出し／書込みのシステムコールを行った場合に，読出し／書込みは行われずALレジスタに02Hが返される．したがって，DTAはバッファの終わりがDS：FFFFHまでに収まるように設定する必要がある．

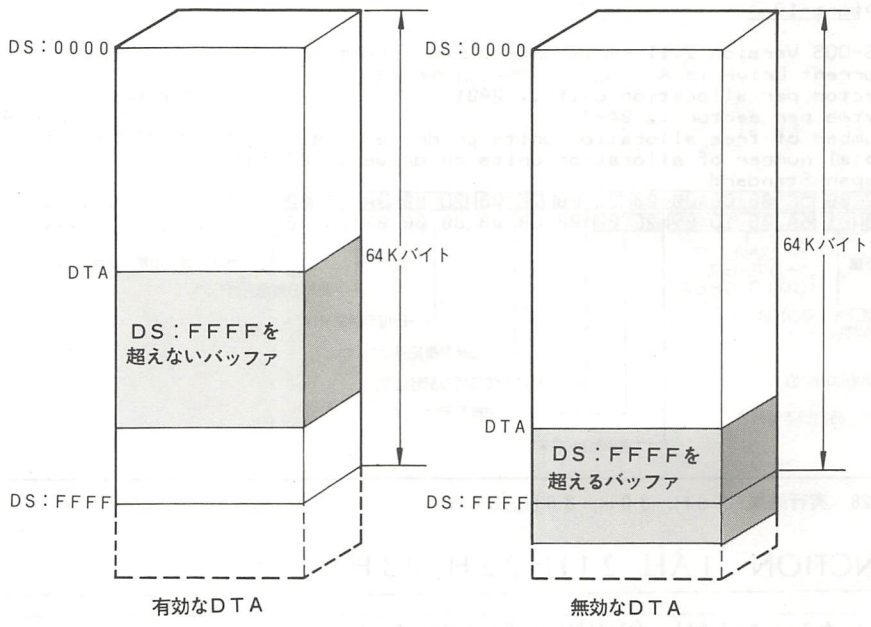


Fig. 7 .29　ディスク転送アドレス

ファンクション21H　(Random Read)——ランダムな読出し

コール手順

```
AH        ←21H
DS：DX    ←オープンされたFCBの先頭アドレス
INT   21H
```

Return

AL＝00H　ファイルからのデータの読出しは正しく行われ，DTAにレコードサイズ(FCBのオフセット0EH)で指定されたバイト数が読み込まれた

AL＝01H　ファイルの終わり．このレコードにデータは存在しない

AL＝02H　DTAに1レコードを読み込む十分な領域がないため，読出しは中止された

AL＝03H　ファイルの終わりまでデータを読み出したが，それが1レコードの大きさに満たなかったため，レコードの残りの部分は00Hで埋められた

解説

相対レコード・フィールド(FCBのオフセット21H)で指定されたレコードがDTAに読み込まれる．カレントブロック(FCBのオフセット0CH)とカレントレコード(FCBのオフセット20H)の各フィールドが相対レコード・フィールドと一致するようにセットされ，次にこれらのフィールドで指定されたレコードがDTAに書き込まれる．

なお，シーケンシャルな読出し(ファンクション14H)の場合と異なり，レコード番号を示す各フィールドはインクリメントされない．

ファンクション22H　(Random Write)——ランダムな書込み

コール手順

```
AH      ← 22H
DS：DX  ←オープンされたFCBの先頭アドレス
INT  21H
```

Return

AL＝00H　書込みは正しく行われた

AL＝01H　ディスクに空き領域がない

AL＝02H　DTAに1レコードを書き込む十分な領域がなく，書込みは中止された

解説

相対レコード・フィールド(FCBのオフセット21H)で指定されたレコードにDTAからデータが書き込まれる．

カレントブロック(FCBのオフセット0CH)とカレントレコード(FCBのオフセット20H)の各フィールドが，相対レコード・フィールドと一致するようにセットされ，次にこれらのフィールドで指定されたレコードにDTAからデータが書き込まれる．

なお，シーケンシャルな書込み(ファンクション15H)の場合と異なり，レコード番号を示す各フィールドはインクリメントされない．

ファンクション23H　(File Size)——ファイルサイズの計算

コール手順

```
AH      ← 23H
DS：DX  ← FCBの先頭アドレス
INT  21H
```

Return

AL＝00H　FCBで指定されたディレクトリ・エントリが存在し，相対レコード・フィールド(FCBのオフセット21H)がセットされた

AL＝FFH　FCBで指定されたディレクトリ・エントリが存在しない

解説

このシステムコールは，ファイルの最終レコードを相対レコード・フィールドにセットする．

FCB内のファイル名と一致するディレクトリ・エントリが見つかると，相対レコード・フィールドから4バイトの領域に，ファイルの最終レコードをセットする．最終レコードは，ディレクトリ・エントリのファイルサイズとFCB内のレコードサイズ(FCBのオフセット0EH)から計算され，次の式を満たす最小のrsとなる．

rs＊(レコードサイズ)≧(ファイルサイズ)

このシステムコールに続けてランダムな書込み(ファンクション22Hまたは28H)を行えば，ちょうどファイルの最後から新しいデータを書き込むことができる．

このシステムコールを行う前に，レコードサイズ・フィールドを実際にファイルの書込みを行ったときと同じレコードサイズにセットしなければならない．そうしないとファイルの終わりのレコードには途中までしかデータがはいっていない場合があり得る．

FUNCTION　1AH, 21H, 22H, 23H　　Sample Program

レコード単位でファイルの順序を逆にするプログラムを作成する．

1レコードは16バイトとし，ファイルサイズを求めて(ファンクション23H)1レコードずつ交換する．

```
                        ;-----------------------------------------;
                        ;   Function  1ah: Set Disk Transfer Address  ;
                        ;             21h: Random Read              ;
                        ;             22h: Random Write             ;
                        ;             23h: File Size                ;
                        ;-----------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
005C                            org     5ch
                        ;
005C    25 [            fcb     db      37 dup (?)
            ??
               ]

                        ;
0100                            org     100h
                        ;
0100  B4 0F             start:  mov     ah,0fh          ┐順序を逆にしようとするファイル
0102  BA 005C R                 mov     dx,offset fcb   │をオープンする．FCBはデフォ
0105  CD 21                     int     21h             │ルトFCBを使用する．ファイル
0107  3C FF                     cmp     al,0ffh         │がない場合は"error"にジャンプ
0109  74 6A                     jz      error           ┘する

010B  C6 06 006A R 10           mov     fcb[14],16 ……レコードサイズを16バイトに設定する

0110  B4 23                     mov     ah,23h          ┐"ファイルサイズの計算"のシ
0112  BA 005C R                 mov     dx,offset fcb   │ステムコール，ファイル・サイズ
0115  CD 21                     int     21h             ┘がレコード数で，FCBの先頭アドレス＋21Hから4バイトの領域にセットされる
```

```
0117  8B 0E 007D R                    mov     cx,word ptr fcb[33]
011B  8A D9                           mov     bl,cl
011D  FE CB                           dec     bl
011F  32 FF                           xor     bh,bh
0121  D1 E9                           shr     cx,1
0123  E3 47                           jcxz    return

0125  B4 1A                     revloop:mov     ah,1ah
0127  BA 0188 R                        mov     dx,offset buffer1
012A  CD 21                           int     21h

012C  88 1E 007D R                    mov     fcb[33],bl
0130  B4 21                           mov     ah,21h
0132  BA 005C R                       mov     dx,offset fcb
0135  CD 21                           int     21h

0137  B4 1A                           mov     ah,1ah
0139  BA 0198 R                       mov     dx,offset buffer2
013C  CD 21                           int     21h

013E  88 3E 007D R                    mov     fcb[33],bh
0142  B4 21                           mov     ah,21h
0144  BA 005C R                       mov     dx,offset fcb
0147  CD 21                           int     21h

0149  88 1E 007D R                    mov     fcb[33],bl
014D  B4 22                           mov     ah,22h
014F  BA 005C R                       mov     dx,offset fcb
0152  CD 21                           int     21h

0154  B4 1A                           mov     ah,1ah
0156  BA 0188 R                       mov     dx,offset buffer1
0159  CD 21                           int     21h

015B  88 3E 007D R                    mov     fcb[33],bh
015F  B4 22                           mov     ah,22h
0161  BA 005C R                       mov     dx,offset fcb
0164  CD 21                           int     21h

0166  FE CB                           dec     bl
0168  FE C7                           inc     bh
016A  E2 B9                           loop    revloop

016C  B4 10                     return: mov     ah,10h
016E  BA 005C R                       mov     dx,offset fcb
0171  CD 21                           int     21h
0173  CD 20                           int     20h

0175  B4 09                     error:  mov     ah,9
0177  BA 017E R                       mov     dx,offset msgerr
017A  CD 21                           int     21h
017C  CD 20                           int     20h

                                ;
017E  0D 0A 45 72 72 6F         msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
0188     10 [                   buffer1 db      16 dup (?)
               ??
                  ]

0198     10 [                   buffer2 db      16 dup (?)
               ??
                  ]


                                ;
01A8                            cseg    ends
                                ;
                                        end     start
```

交換するレコード番号をBLレジスタとBHレジスタにセットする

レコード数が1以下なら交換する必要がないので"return"へジャンプする

DTAを"buffer 1"の先頭にセットして，BL番目のレコードを"buffer 1"に読み込む

DTAを"buffer2"の先頭にセットして，BH番目のレコードを"fuffer 2"に読み込む

"buffer 2"に読込んだデータをBL番目のレコードに書き込む

DTAを"buffer1"の先頭にセットして、"buffer1"に読込んだデータをBH番目のレコードに書き込む

次のレコードを交換する

"revloop"へ戻る

順序の入換えの終わったファイルをクローズしてプログラムを終了する

エラーメッセージを表示してプログラムを終了する

Fig. 7.30 Sample List 1AH, 21H, 22H, 23H

実行手順　　FUNC－1A　〈ファイル名〉 ↵

```
A>dump test ↵ ……"TEST" をDUMPで表示する

Dump Version 2.0

00000000  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
00000010  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
00000020  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
00000030  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
00000040  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
00000050  50 51 52 53 54 55 56 57-58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[¥]^_
00000060  60 61 62 63 64 65 66 67-68 69 6A 6B 6C 6D 6E 6F   `abcdefghijklmno
00000070  70 71 72 73 74 75 76 77-78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
00000080  80 81 82 83 84 85 86 87-88 89 8A 8B 8C 8D 8E 8F   ................
00000090  90 91 92 93 94 95 96 97-98 99 9A 9B 9C 9D 9E 9F   ................

A>func-1a test ↵ ……"FUNC-1A.COM" を起動する

A>dump test ↵ ……順序を逆転した "TEST" を再び表示

Dump Version 2.0 ……16バイト単位で逆順に並べ換えられている

00000000  90 91 92 93 94 95 96 97-98 99 9A 9B 9C 9D 9E 9F   ................
00000010  80 81 82 83 84 85 86 87-88 89 8A 8B 8C 8D 8E 8F   ................
00000020  70 71 72 73 74 75 76 77-78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{|}~.
00000030  60 61 62 63 64 65 66 67-68 69 6A 6B 6C 6D 6E 6F   `abcdefghijklmno
00000040  50 51 52 53 54 55 56 57-58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[¥]^_
00000050  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
00000060  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
00000070  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
00000080  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
00000090  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................

A>■
```

Fig. 7.31　実行結果　1AH，21H，22H，23H

FUNCTION　24H, 27H, 28H, 29H

ファンクション24H　(Set Relative Record)――相対レコード・フィールドの設定

コール手順

AH　　←24H
DS：DX　←オープンされたFCBの先頭アドレス
INT　21H

Return

なし

解説

ランダム・アクセスでは，相対レコード・フィールドの値によってどのレコードに読出し／書込みを行うかを決め，シーケンシャル・アクセスではカレントブロックとカレントレコードの2

つのフィールドの値から，次の式で与えられるレコードに読出し／書込みを行う．

（カレントブロック）＊128＋（カレントレコード）

そこでシーケンシャル・アクセスで読出しまたは書込みを行ったレコードの次のレコードからランダム・アクセスを行おうとする場合に，このシステムコールによって相対レコードをセットする．

なお，ランダム・アクセスを行った後は，相対レコード，カレントブロック，カレントレコードのすべてのフィールドが更新されるので，ランダム・アクセス後のシーケンシャル・アクセスでは，特に値を設定する必要はない．

ファンクション 27H　(Random Block Read)——ランダムなブロック読出し

コール手順

```
AH      ←27H
DS：DX  ←オープンされたFCBの先頭アドレス
CX      ←読み出すレコード数
INT  21H
```

Return

AL＝00H　読出しは正常に行われた

AL＝01H　ファイルの終わり(EOF)．このレコードにデータは存在しない

AL＝02H　DTAに1レコードを読み込むための領域がなく，読込みは中止された

AL＝03H　EOFまでファイルを読んだが1レコードに達しなかった．レコードの残りの部分が00Hで埋められた

CX←読み出されたレコード数

解説

CXレジスタで指定されたレコード数のデータが，ファイルからDTAに読み込まれる．ファイルからの読出しはFCBの相対レコード・フィールド(FCBのオフセット21H)で指定されたレコードから開始され，CXレジスタに読み出されたレコード数が返される．

カレントブロック(FCBのオフセット0CH)，カレントレコード(FCBのオフセット20H)および相対レコード(FCBのオフセット21H)の各フィールドは，読み出した次のレコードを示すようにセットされる．

ファンクション21Hでは，1レコード単位でしか読み出せなかったが，このシステムコールでは連続した複数個のレコードが読み出せる．

実際にDTAに読み込んだレコード数は，CXレジスタに返される．このとき，CXレジスタの

値が指定したレコード数より小さな場合は，最後のレコードを読み込んだところでファイルが終了したか，次にレコードを読み込むとDTAの領域がDS：FFFFHを超えてしまうかのどちらかであることを意味する．この場合でもカレントブロック，カレントレコードおよび相対レコードの各フィールドは，読み出した最後のレコードを示すようにセットされる．

ファンクション28H　(Random Block Write)——ランダムなブロック書込み

コール手順

```
AH      ←28H
DS：DX  ←オープンされたFCBの先頭アドレス
CX      ←書き込むレコード数．0000Hの場合，FCBの相対レコード・フィールド(FCB
          のオフセット21H)の値と同じレコード数がディレクトリ・エントリのファイ
          ルサイズ・フィールド(オフセット1CH)にセットされる
INT　21H
```

Return

AL＝00H　書込みが正常に行われた

AL＝01H　ディスクに空き領域がない

AL＝02H　DTAに1レコードを書き込むだけのデータ領域がないため，書込みは中止された

CX←書き込まれたレコード数

解説

CXレジスタで指定されたレコード数のデータが，DTAからファイルに書き込まれる．ファイルへのレコードの書込みは，相対レコード・フィールドで指定されたレコードから開始される．

CX＝0000Hの場合は，ディスクへの書込みは行われないが，相対レコード・フィールドで指定された値にファイルの大きさがセットされる．

CXレジスタには，書き込まれたレコード数が返され，カレントブロック(FCBのオフセット0CH)，カレントレコード(FCBのオフセット20H)および相対レコード(FCBのオフセット21H)の各フィールドの値は，書き込んだ次のレコードを示すようにセットされる．

ファンクション22Hでは1レコード単位でしか書き込めないが，このシステムコールでは連続した複数個のレコードが書き込める．

CXレジスタで指定したレコード数を書き込むとディスクフルになる場合には，レコードはまったく書き込まれずCXレジスタに0000H，ALレジスタに01Hが返される．また次のレコードを書き込む際にDS：FFFFHを超えるレコードを書き込もうとすると，そこで書込みは中止され，CXレジスタにそれまでに書き込まれたレコード数，ALレジスタに02Hがセットされ，書き

込んだ次のレコードを示すようにカレントブロック，カレントレコードおよび相対レコードの各フィールドがセットされる．

ファンクション 29H　(Parse File Name)——ファイル名の解析

コール手順

```
AH      ← 29H
AL      ←解析の制御
DS：SI  ←解析する文字列の先頭アドレス
ES：DI  ←オープンされていない FCB の先頭アドレス
INT  21H
```

Return

AL＝00H　ワイルドカードが使用されていない

AL＝01H　ワイルドカードが使用されている

AL＝FFH　ドライブ名が無効である

DS：SI←解析された文字列の中のファイル名の直後のアドレス

解説

DS：SI で指定されたアドレスからはじまるファイル名(〝d：ファイル名．拡張子〞)を，ES：DI で指定したアドレスにオープンされていない FCB の形式でセットする．ファンクション 0AH のシステムコールやコマンドラインにセットされたファイル名を FCB にセットして，ファイルのオープンなどを行うときに有効である．

ファイル名または拡張子にアスタリスク〝＊〞がはいっている場合には，それ以後 FCB 内のファイル名または拡張子の部分には〝？〞が埋められる．

文字列の中にワイルドカード(〝＊〞または〝？〞)があった場合には AL レジスタに 01H を返し，ドライブ名が無効な場合は AL レジスタに FFH を返す．

ファイル名の区切り記号には

：　．　；　，　＝　＋　／　〝　[　]　¥　<　>　|　スペース　タブ

があり，ファイル名の終わりは，ファイル名区切り記号とすべての制御文字(00H～1FH のコード)で示される．このシステムコールはファイル名の終わりに達したところで解析を終了し，そのアドレスを DS：SI に返す．制御文字が含まれる終了記号を検出した時点で解析は停止するので，ファイル名にファイル名の終了記号を入れることはできない．

AL レジスタのビット 0～3 により，解析処理を制御できる．ビット 4～7 は無視される．Table 7.2 に 0～3 の各ビットの意味を示す．

Table 7.2　ファイル名解析の制御ビット

ビット	値	機　能
0	0	ファイル区切り記号を検出した場合，すべての解析を停止させる
	1	先行するファイル区切り記号は無視する
1	0	文字列にドライブ番号がはいっていない場合，FCBのドライブ番号(FCBのオフセット００H)には００H(カレント・ドライブ)がセットされる
	1	文字列にドライブ番号がはいっていない場合，FCBのドライブ番号は変更されない
2	0	文字列にファイル名がはいっていない場合，FCBのファイル名には８つのスペースがセットされる
	1	文字列にファイル名がはいっていない場合，FCBのファイル名は変更されない
3	0	文字列に拡張子がはいっていない場合，FCBの拡張子には３つのスペースがセットされる
	1	文字列に拡張子がはいっていない場合，FCB内の拡張子は変更されない

FUNCTION　24H, 27H, 28H, 29H　　Sample Program

ランダムなブロック読出し／書込みを用いてファイルの連結を行うプログラムを作成する．ソースファイルをディスティネーションファイルの後ろに連結したファイルをディスティネーションファイルとする．このとき，ソースファイルとディスティネーションファイルはテキストファイルであるとし，ファイルの最後は１個のCTRL-Z(1AH)で終わっているものとする．

```
                                ;-----------------------------------------;
                                ;   Function  24h: Set Relative Record    ;
                                ;             27h: Random Block Write     ;
                                ;             28h: Random Block Read      ;
                                ;             29h: Parse File Name        ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
005C                                    org     5ch
                                ;
005C    25 [                    fcb     db      37 dup (?)
            ??
               ]

                                ;
0100                                    org     100h
                                ;
0100  BA 01E9 R                 start:  mov     dx,offset prompt1
0103  E8 01B2 R                         call    getname
0106  75 0B                             jnz     toerr
0108  B4 0F                             mov     ah,0fh
010A  BA 005C R                         mov     dx,offset fcb
010D  CD 21                             int     21h
010F  3C FF                             cmp     al,0ffh
0111  75 03                             jne     noterr
0113  E9 01A9 R                 toerr:  jmp     error
```

ソースファイルを入力しオープンする
入力した文字列を"fcb"にセットするために"getname"をコールする
ファイルがないか，あるいはオープンできない場合は"error"へジャンプする

```
0116  B4 1A                    noterr: mov     ah,1ah
0118  BA 0204 R                       mov     dx,offset buffer
011B  CD 21                           int     21h

011D  C7 06 006A R 0001               mov     word ptr fcb[0eh],1
0123  C6 06 007C R 00                 mov     fcb[20h],0
0128  B4 24                           mov     ah,24h
012A  BA 005C R                       mov     dx,offset fcb
012D  CD 21                           int     21h

012F  A1 006C R                       mov     ax,word ptr fcb[10h]
0132  A3 0201 R                       mov     num_rec,ax

0135  B4 27                           mov     ah,27h
0137  BA 005C R                       mov     dx,offset fcb
013A  8B 0E 0201 R                    mov     cx,num_rec
013E  CD 21                           int     21h

0140  BA 01F5 R                       mov     dx,offset prompt2
0143  E8 01B2 R                       call    getname
0146  75 61                           jnz     error
0148  B4 0F                           mov     ah,0fh
014A  BA 005C R                       mov     dx,offset fcb
014D  CD 21                           int     21h
014F  3C FF                           cmp     al,0ffh
0151  74 56                           je      error

0153  B4 1A                           mov     ah,1ah
0155  BA 0203 R                       mov     dx,offset readbf
0158  CD 21                           int     21h

015A  C7 06 006A R 0001               mov     word ptr fcb[0eh],1
0160  C6 06 007C R 00                 mov     fcb[20h],0

0165  B4 14                    readlp: mov     ah,14h
0167  BA 005C R                       mov     dx,offset fcb
016A  CD 21                           int     21h
016C  3C 00                           cmp     al,0
016E  75 17                           jne     endfl
0170  80 3E 0203 R 1A                 cmp     readbf,1ah
0175  75 EE                           jne     readlp

0177  80 2E 007C R 01                 sub     fcb[20h],1
017C  73 09                           jnc     endfl
017E  FF 0E 0068 R                    dec     word ptr fcb[0ch]
0182  C6 06 007C R 7F                 mov     fcb[20h],7fh

0187  B4 1A                    endfl:  mov     ah,1ah
0189  BA 0204 R                       mov     dx,offset buffer
018C  CD 21                           int     21h

018E  B4 24                           mov     ah,24h
0190  BA 005C R                       mov     dx,offset fcb
0193  CD 21                           int     21h

0195  B4 28                           mov     ah,28h
0197  BA 005C R                       mov     dx,offset fcb
019A  8B 0E 0201 R                    mov     cx,num_rec
019E  CD 21                           int     21h

01A0  B4 10                           mov     ah,10h
01A2  BA 005C R                       mov     dx,offset fcb
01A5  CD 21                           int     21h
01A7  CD 20                           int     20h

01A9  B4 09                    error:  mov     ah,9
01AB  BA 01DF R                       mov     dx,offset msgerr
01AE  CD 21                           int     21h
01B0  CD 20                           int     20h
```

DTAを"buffer"にセットする

レコードサイズを1バイトとし相対レコード・フィールドをファイルの先頭にセットする

ソースファイルをオープンしたときセットされるファイルの大きさを"num_rec"に退避する

ソースファイルの大きさ分"buffer"にデータを読み込む

ディスティネーションファイルを入力し，オープンする
ファイル名は，"getname"内でFCBにセットされる
ファイルがないか，あるいはオープンできない場合は"error"へジャンプする

DTAを"readbf"にセットする

レコードサイズを1バイトにセットし，カレントレコード・フィールドを0にする

ファイルの終わりかCTRL－Z（1AH）がくるまでディスティネーションファイルを読み込む
ファイルの終わりの場合は"endfl"へジャンプする

CTRL－Z(1AH)ならばレコード番号を1つ前に戻す

DTAを"buffer"にセットする

相対レコード・フィールドをディスティネーションファイルの最後にセットする

ディスティネーションファイルの最後に前に読んだリースファイルのデータを書き込む

ディスティネーションファイルをクローズし，プログラムを終了する

エラーメッセージを表示しプログラムを終了する

```
01B2  B4 09                     getname:mov     ah,9
01B4  CD 21                             int     21h
01B6  B4 0A                             mov     ah,0ah
01B8  BA 01CE R                         mov     dx,offset flname
01BB  CD 21                             int     21h
01BD  B4 29                             mov     ah,29h
01BF  B0 0F                             mov     al,0fh
01C1  BE 01D0 R                         mov     si,offset flname[2]
01C4  BF 005C R                         mov     di,offset fcb
01C7  1E                                push    ds
01C8  07                                pop     es
01C9  CD 21                             int     21h
01CB  84 C0                             test    al,al
01CD  C3                                ret
                                ;
01CE  0F 10 [                   flname  db      15,16 dup (?)
              ??
                  ]

01DF  0D 0A 45 72 72 6F         msgerr  db      0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
01E9  0D 0A 53 6F 75 72         prompt1 db      0dh,0ah,'Source ? $'
      63 65 20 3F 20 24
01F5  0D 0A 20 20 44 65         prompt2 db      0dh,0ah,'  Dest ? $'
      73 74 20 3F 20 24
0201  ????                      num_rec dw      ?
0203  ??                       readbf  db      ?
0204 * 8000 [                   buffer  db      32768 dup (?)
              ??
                  ]

                                ;
8204                            cseg    ends
                                ;
                                        end     start
```

"finame"に入力された名前を解析してFCBにセットするサブルーチン
名前解析の機能は0FHに設定しているので，省略した部分は前と同じものが採用される

Fig. 7.32　Sample List　24H, 27H, 28H, 29H

実行手順　　FUNC－24↵

```
A>type fat.3↵ ……"FAT.3" の内容をタイプしてみる
3.5  FILE ALLOCATION TABLE (FAT)

     The following  information is included for system programmers who wish to
write installable device drivers.   This section explains  how MS-DOS uses the
File ... to convert the  clusters of a file  to logical sect...

disk.  Programs must use the ... access-
ing files;  programs that access the FAT  are not guaranteed to  be  upwardly-
compatible with future releases of MS-DOS.


A>type fat.4↵ ……"FAT.4" の内容をタイプしてみる
     The File Allocation Table  is  an array of 12-bit entries (1.5 bytes) for
each cluster  on the disk.   The first two FAT  entries  map  a portion of the
directory. The FAT entries indicate the size and format of the disk.

the next cluster allocated, regardless of ... on the disk.
This permits  the most efficient  utilization  of disk space  because clusters
made available by erasing files can be allocated for new files.
```

```
A>func-24 fat.4 ……"FUNC-24．COM"を実行する."FAT．4" は "ファイル名の解析" のシステムコールのデフォルト値
                    としてFCBにセットする
Source ? fat ……拡張子を省略したのでソースファイルは "FAT．4" となる
  Dest ? fat.3 ……ディスティネーションファイルは "FAT．3" を指定する
A>type fat.3 ……"FAT．3" の後ろに "FAT．4" の内容がつながって "FAT．3" となっていることを確認する
3.5  FILE ALLOCATION TABLE (FAT)

     The following  information is included for system programmers who wish to
write installable device drivers.   This section explains  how MS-DOS uses the
                                                                    .

ing files;  programs that access the ...                        upwardly-
compatible with future releases of MS-DOS.
      ここでファイルが結合されている
     The File Allocation Table  is  an array of 12-bit entries (1.5 bytes) for
each cluster  on the disk.   The first two FAT  entries  map  a portion of the
                                  indicate the size and format of the disk.

the next cluster allocated,  regardless of ...                  on the disk.
This permits  the most efficient  utilization  of disk space  because clusters
made available by erasing files can be allocated for new files.

A>■
```

Fig. 7.33　実行結果　24H, 27H, 28H, 29H

FUNCTION　25H, 35H

ファンクション 25H　(Set Vector)——割込みベクトルの設定

コール手順

AH	←25H
AL	←割込みベクトル番号
DS：DX	←割込み処理ルーチンのエントリ・アドレス
INT　21H	

Return

なし

解説

このファンクションは，特定の割込みベクトルを設定するために用いる．MS-DOS はこのシステムコールによってプロセスごとに割込みを管理しており，内部割込み 22H～24H のベクトルをプロセスの実行時に設定し，終了アドレスや，致命的エラー，CTRL-C による割込みが発生したときの処理を管理する．

ファンクション 35H　(Get Interrupt Vector)——割込みベクトルの読出し

コール手順

AH　　← 35H
AL　　←割込みベクトルの番号
INT　21H

Return

ES：BX←割込みベクトル

解説

割込みベクトルのセグメントをESレジスタにオフセットをBXレジスタに返す.

FUNCTION　25H, 35H　　Sample Program

割込みベクトルを設定し，それを読み出して表示するプログラムを作成する．割込みベクトルとその番号はプログラム中で与える.

```
                                ;-----------------------------------------;
                                ;   Function  25h: Set Vector             ;
                                ;             35h: Get Interrupt Vector   ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        extrn   hexout:near ……"hexout"を外部サブルーチンとして定義する
                                        assume  cs:cseg,ds:cseg
                                ;
= 0080                          interpt equ     80h
= 1000                          seg_adr equ     1000h
= 1200                          off_adr equ     1200h
                                ;
0100                                    org     100h
                                ;
0100  1E                        start:  push    ds
0101  B8 1000                          mov     ax,seg_adr
0104  8E D8                             mov     ds,ax
0106  BA 1200                          mov     dx,off_adr      内部割込み番号80Hのベクトルを1000：1200にセットする
0109  B0 80                             mov     al,interpt
010B  B4 25                             mov     ah,25h
010D  CD 21                             int     21h
010F  1F                                pop     ds

0110  B4 09                             mov     ah,9
0112  BA 012F R                         mov     dx,offset msgint
0115  CD 21                             int     21h
0117  B4 35                             mov     ah,35h
0119  B0 80                             mov     al,interpt      内部割込み番号80Hのベクトルを取り出して表示する
011B  CD 21                             int     21h             これがセットした値と同じであることを確かめる
011D  8C C0                             mov     ax,es
011F  E8 0000 E                         call    hexout
0122  B4 02                             mov     ah,2
0124  B2 3A                             mov     dl,':'
0126  CD 21                             int     21h
0128  8B C3                             mov     ax,bx
012A  E8 0000 E                         call    hexout

012D  CD 20                             int     20h……プログラムの終了
```

```
                                        ;
012F  0D 0A 56 65 63 74                 msgint  db  0dh,0ah,'Vector of Interrupt No. 80
                                        H is $'
      6F 72 20 6F 66 20
      49 6E 74 65 72 72
      75 70 74 20 4E 6F
      2E 20 38 30 48 20
      69 73 20 24
                                        ;
0151                                    cseg    ends
                                        ;
                                                end     start
```

Fig. 7.34 Sample List 25H, 35H

実行手順　　FUNC−25↵

```
A>func-25 ↵……"FUNC-25.COM" の実行

Vector of Interrupt No. 80H is 1000:1200 ……内部割込み80Hのベクトルに1000：1200が書
A>■                                        き込まれたことがわかる
```

Fig. 7.35 実行結果 25H, 35H

割込みベクトルとその番号を変えて実行してみてください．このように割込みベクトルを変えて自分用のシステムコールを作成することもできます．ただし，MS-DOS の使用している割込みベクトルを変更すると結果は保証されません．

FUNCTION 2EH, 54H

ファンクション 2EH (Set／Reset Verify Flag)——ベリファイフラグの設定

コール手順

AH ←2EH
AL ←ビット 0：0 ベリファイを行わない
　　：1 ベリファイを行う
INT 21H

Return

なし

解説

ベリファイフラグの ON／OFF に利用する．

ベリファイフラグを ON にすると，MS-DOS はディスクにデータを書き込むごとに読出しを行ってその内容の検査を行う．重要なデータをディスクに書き込む場合は ON にした方がよい．

ディスクエラーはまれであり，ベリファイを行うとディスクへの書込みに時間がかかるので通常は OFF にしておいてよい．

ファンクション54H　(Return Current Setting of Verify After Write Flag)
——ベリファイフラグの読出し

コール手順

AH　←54H
INT　21H

Return

AL← 現在のベリファイフラグの値(0：OFF／1：ON)

解説

現在のベリファイフラグがOFFならば　AL＝00H，ONならばAL＝01Hとなる

FUNCTION　2EH, 54H　　Sample Program

ベリファイフラグを設定したり，現在のベリファイフラグの状態を返すVERIFYコマンドに相当するプログラムを作成する．

```
                                ;-----------------------------------------;
                                ;   Function  2eh: Set/Reset Verify Flag  ;
                                ;             54h: Return Curent Setting ;
                                ;                  of Verify Flag       ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0080                                    org     80h
                                ;
0080    80 [                    cmdln   db      128 dup (?)
            ??
                ]

                                ;
0100                                    org     100h
                                ;
0100  A0 0082 R                 start:  mov     al,cmdln[2]     ; 与えられたパラメータが
0103  3C 30                             cmp     al,'0'          ; "0" または "1" なら
0105  74 04                             je      flgset          ; ばベリファイフラグのセ
0107  3C 31                             cmp     al,'1'          ; ット／リセットを行う
0109  75 04                             jne     flgget          ; それ以外なら"flgget"
                                                                ; へジャンプする
010B  B4 2E                     flgset: mov     ah,2eh          ; "ベリファイフラグの設
010D  CD 21                             int     21h             ; 定のシステムコール
010F  B4 09                     flgget: mov     ah,9            ;
0111  BA 012A R                         mov     dx,offset msgvrfy
0114  CD 21                             int     21h
0116  B4 54                             mov     ah,54h          ; ベリファイフラグがON
0118  CD 21                             int     21h             ; ならば
011A  BA 013C R                         mov     dx,offset msgon ; "Verify Flag is on."
011D  3C 01                             cmp     al,1            ; OFFならば
011F  74 03                             je      vrfyflg         ; "Verify Flag is off."
0121  BA 0142 R                         mov     dx,offset msgoff; と表示する
0124  B4 09                     vrfyflg:mov     ah,9            ;
0126  CD 21                             int     21h             ;
```

```
0128  CD 20                              int     20h ……… プログラムの終了
                                    ;
012A  0D 0A 56 65 72 69             msgvrfy db  0dh,0ah,'Verify Flag is $'
      66 79 20 46 6C 61
      67 20 69 73 20 24
013C  6F 6E 2E 0D 0A 24             msgon   db  'on.',0dh,0ah,'$'
0142  6F 66 66 2E 0D 0A             msgoff  db  'off.',0dh,0ah,'$'
      24
                                    ;
0149                                cseg    ends
                                    ;
                                            end     start
```

Fig. 7.36 Sample List 2EH, 54H

実行手順　FUNC−2E↵

現在のベリファイフラグの状態を返す

FUNC−2E {0|1}↵

ベリファイフラグを { 0…OFF / 1…ON } に設定する

```
A>verify↵
VERIFY は <OFF> です          } 現在ベリファイフラグはOFFとなっている

A>func-2e↵                   } "FUNC-2E.COM" をパラメータなしで実行すると
                             } ベリファイフラグのON/OFFを表示する
Verify Flag is off.

A>func-2e 1↵                 } パラメータに "1" を与えてベリファイフラグをONにして
                             } これをVERIFYコマンドで確認する
Verify Flag is on.

A>verify↵
VERIFY は <ON> です

A>func-2e 0↵                 } パラメータに "0" を与えてベリファイフラグをOFFにして
                             } これをVERIFYコマンドで確認する
Verify Flag is off.

A>verify↵
VERIFY は <OFF> です

A>■
```

Fig. 7.37 実行結果 2EH, 54H

FUNCTION　33H

ファンクション 33H　(CTRL-C Check)——ブレークチェックの制御

コール手順

ブレークチェックの設定を得る場合	ブレークチェックを設定する場合
AH　←33H	AH　←33H
AL　←00H	AL　←01H
INT　21H	DL　←ビット0：0　ブレークチェックをOFF
	：1　ブレークチェックをON
	INT　21H

Return

AL＝FFH　ALレジスタの値が00H～01Hの範囲外である

それ以外　設定を得る場合には｛OFFならばDL＝00H / ONならばDL＝01H｝が返される

解説

ブレークチェックがOFFのときにはMS-DOSは01H～0CH(06H, 07Hを除く)までのシステムコールを実行しているときのみブレークチェックを行うが，ブレークチェックがONのときにはすべてのシステムコールに対してこのチェックを行う．たとえば，ブレークチェックがOFFの場合はすべてのディスク・アクセスは割込みがかけられずに続行するが，ONの場合はディスク・アクセスを開始したシステムコールにおいてもCTRL-Cで割込みがかけられる．

FUNCTION　33H　Sample Program

ブレークチェックをセット／リセットしたり，現在のブレークチェックの状態を返すBREAKコマンドに相当するプログラムを作成する．

```
                        ;---------------------------------------;
                        ;   Function  33h: CTRL-C Check         ;
                        ;---------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
0080                            org     80h
                        ;
0080    80 [            cmdln   db      128 dup (?)
           ??
               ]
                        ;
0100                            org     100h
```

```
0100  8A 16 0082 R                ;
0104  80 FA 30                    start:  mov     dl,cmdln[2]
0107  74 05                               cmp     dl,'0'
0109  80 FA 31                           je      flgset
010C  75 06                               cmp     dl,'1'
                                          jne     flgget

010E  B4 33                       flgset: mov     ah,33h
0110  B0 01                               mov     al,1
0112  CD 21                               int     21h

0114  B4 09                       flgget: mov     ah,9
0116  BA 0135 R                           mov     dx,offset msgctlc
0119  CD 21                               int     21h
011B  B4 33                               mov     ah,33h
011D  B0 00                               mov     al,0
011F  CD 21                               int     21h
0121  80 FA 01                           cmp     dl,1
0124  74 06                               je      flg_on
0126  BA 0153 R                           mov     dx,offset msgoff
0129  EB 04 90                            jmp     next

012C  BA 014D R                   flg_on: mov     dx,offset msgon

012F  B4 09                       next:   mov     ah,9
0131  CD 21                               int     21h

0133  CD 20                              int     20h
                                  ;
0135  0D 0A 43 54 52 4C           msgctlc db  0dh,0ah,'CTRL-C Check Flag is $'
      2D 43 20 43 68 65
      63 6B 20 46 6C 61
      67 20 69 73 20 24
014D  6F 6E 2E 0D 0A 24           msgon   db  'on.',0dh,0ah,'$'
0153  6F 66 66 2E 0D 0A           msgoff  db  'off.',0dh,0ah,'$'
      24
                                  ;
015A                              cseg    ends
                                  ;
                                          end     start
```

与えられたパラメータが"0"または"1"ならブレークチェックのセット／リセットを行う
それ以外なら"flgget"へジャンプする

"ブレークチェックの制御"の"セット／リセット"のシステムコール

ブレークチェックがONならば"CTRL-C Check Flag is on."
OFFならば"CTRL-C Check Flag is off."
と表示する

プログラムの終了

Fig. 7.38 Sample List 33H

実行手順　FUNC−33↵

現在のブレークチェックの状態を返す

FUNC−33 {0|1}↵

ブレークチェックを $\left\{\begin{matrix}0\cdots\text{OFF}\\1\cdots\text{ON}\end{matrix}\right\}$ に設定する

```
A>break↵
BREAK は <OFF> です                現在，ブレークチェックはOFFとなっている

A>func-33↵
CTRL-C Check Flag is off.         FUNC-33.COMをパラメータなしで実行するとブレークチェックのON／OFFを表示する

A>func-33 1↵
CTRL-C Check Flag is on.          パラメータに"1"を与えてブレークチェックをONにする
                                  これをBREAKコマンドで確認する
A>break↵
BREAK は <ON> です
```

```
A>func-33 0↵
CTRL-C Check Flag is off.

A>break↵
BREAK は <OFF> です

A>■
```

こんどはパラメータに "0" を与えてブレークチェックをOFFにする

Fig. 7.39　実行結果　33H

ブレークチェックをONにして，ファイルを書込み中にCTRL-Cで割込みをかけると，ディスクリセットが行われるためそのファイルの内容は保証されません．

FUNCTION　39H, 3AH, 3BH

ファンクション39H　(Create Sub-Directory)——サブ・ディレクトリの作成

コール手順

AH　　←39H
DS：DX　←パス名の先頭アドレス
INT　21H

Return

キャリーがセットされた場合
AX＝03H　指定されたパス名が無効であるか，またはパスが存在しない
AX＝05H　アクセスが否定された
ルート・ディレクトリに空き領域がないか，すでに同名のファイルまたはディレクトリが存在しているので，ディレクトリを作成することはできない

キャリーがセットされない場合
ディレクトリが作成された

解説

ASCIZ文字列でパス名を指定し，その先頭アドレスをDS：DXにセットしてシステムコールを行えば，パス名で指定したディレクトリ・エントリを作成する．

ファンクション3AH　(Remove a Directory Entry)——サブ・ディレクトリの削除

コール手順

AH　　←3AH
DS：DX　←パス名の先頭アドレス
INT　21H

Return

キャリーがセットされた場合

AX＝03H　指定されたパス名が無効であるか，またはパスが存在しない

AX＝05H　アクセスが否定された

指定されたディレクトリが空でなかったか，ディレクトリでないか，あるいはルート・ディレクトリであった

AX＝10H　カレント・ディレクトリを指定した

キャリーがセットされない場合

ディレクトリが削除された

解説

パス名で示されたディレクトリが削除される．

ASCIZ 文字列で指定したパス名の先頭アドレスを DS：DX にセットしてこのシステムコールを行えば，パス名で指定したディレクトリを削除する．

ファンクション 3BH （Change the Current Directory）——カレント・ディレクトリの変更

コール手順

```
AH      ← 3BH
DS：DX  ←パス名の先頭アドレス
INT  21H
```

Return

キャリーがセットされた場合

AX＝03H　指定されたパス名が無効であるか，またはパスが存在しない

キャリーがセットされない場合

カレント・ディレクトリが変更された

解説

指定されたパス名のディレクトリをカレント・ディレクトリとする．指定されたパス名が無効であれば，カレント・ディレクトリは変更されない．

FUNCTION　39H, 3AH, 3BH　　Sample　Program

まず，FUNC−39 でプログラム中で指定したディレクトリを作成し，そのディレクトリをカレント・ディレクトリとする．次に〝CHDIR．．〟で親ディレクトリに移り FUNC−3A を用いてそのディレクトリを削除してみる．

```
                                ;-----------------------------------------;
                                ;   Function  39h: Create Sub-Directory   ;
                                ;             3bh: Change the Current     ;
                                ;                          Directory      ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
                                ;
0100  B4 39                     start:  mov     ah,39h          ┐"サブ・ディレクトリの作
0102  BA 0132 R                         mov     dx,offset new_dir  成" のシステムコール.
0105  CD 21                             int     21h             │これにより"new-dir"
0107  72 0F                             jc      error           │をカレント・ディレクト
0109  B4 3B                             mov     ah,3bh          │リに作成する
010B  BA 0132 R                         mov     dx,offset new_dir  カレント・ディレクトリ
010E  CD 21                             int     21h             │を新しく作った "new-
0110  72 06                             jc      error           │dir" に変更する
0112  BA 0121 R                         mov     dx,offset msgok ┐
0115  EB 04 90                          jmp     next            ┘"Ok" のメッセージ

0118  BA 0128 R                 error:  mov     dx,offset msgerr ……"Error" のメッセージ

011B  B4 09                     next:   mov     ah,9            ┐
011D  CD 21                             int     21h             ┘メッセージの出力

011F  CD 20                             int     20h ……プログラムの終了
                                ;
0121  0D 0A 4F 6B 0D 0A         msgok   db  0dh,0ah,'Ok',0dh,0ah,'$'
      24
0128  0D 0A 45 72 72 6F         msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
0132  6E 65 77 5F 64 69         new_dir db  'new_dir',0
      72 00
                                ;
013A                            cseg    ends
                                ;
                                        end     start
```

```
                                ;-----------------------------------------;
                                ;   Function  3ah: Remove a Directory Entry ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
                                ;
0100  B4 3A                     start:  mov     ah,3ah          ┐"サブ・ディレクトリの削
0102  BA 0126 R                         mov     dx,offset new_dir  除" のシステムコール
0105  CD 21                             int     21h             │"new-dir" というディ
                                                                │レクトリを削除する
0107  BA 011C R                         mov     dx,offset msgerr ┐
010A  72 03                             jc      error           │削除できれば "Ok", で
010C  BA 0115 R                         mov     dx,offset msgok │きなければ "Error" と
010F  B4 09                     error:  mov     ah,9            │表示する. ディレクトリ
0111  CD 21                             int     21h             │内にファイルやディレク
                                                                │トリが作られていれば,
0113  CD 20                             int     20h ……プログラムの終了 ┘そのディレクトリは削除
                                                                 できない
                                ;
0115  0D 0A 4F 6B 0D 0A         msgok   db  0dh,0ah,'Ok',0dh,0ah,'$'
      24
011C  0D 0A 45 72 72 6F         msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
0126  6E 65 77 5F 64 69         new_dir db  'new_dir',0
      72 00
                                ;
012E                            cseg    ends
                                ;
                                        end     start
```

Fig. 7.40　Sample List　39H, 3AH, 3BH

実行手順　　FUNC－39 ↵
CHDIR　. . ↵
FUNC－3A ↵

```
A>dir new_dir ↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

ファイルが見つかりません
```

"new-dir" というディレクトリまたはファイルが存在しないことを確認する

```
A>func-39 ↵
```

……"FUNC-39.COM" を実行する

```
Ok
```

……正常に終了したメッセージ

```
A>dir ↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥new_dir

.            <DIR>      84-03-22  17:50
..           <DIR>      84-03-22  17:50
       2 個のファイルがあります
  326656 バイトが使用可能です
```

新たに "new-dir" というディレクトリが作成されカレント・ディレクトリが "new-dir" に移っている

```
A>copy ¥func-39.asm ↵
```

……ディレクトリ"new-dir" にファイルを作成する

```
        1 個のファイルをコピーしました

A>chdir .. ↵
```

……親ディレクトリに移る

```
A>dir new_dir ↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥new_dir

.            <DIR>      84-03-22  17:50
..           <DIR>      84-03-22  17:50
FUNC-39  ASM       650  84-02-21  16:15
       3 個のファイルがあります
  325632 バイトが使用可能です
```

作成したディレクトリ "new-dir" の内容を確認する
"FUNC-39.ASM" がコピーされている

```
A>func-3a ↵
```

……"FUNC-3A.COM" を起動する

```
Error
```

……"new-dir" が空でなかったので削除できない

```
A>del new_dir ↵
よろしいですか <Y/N>? y
```

"new-dir" を空にする

```
A>dir new_dir ↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥new_dir

.            <DIR>      84-03-22  17:50
..           <DIR>      84-03-22  17:50
       2 個のファイルがあります
  326656 バイトが使用可能です
```

空になったことを確認する

```
A>func-3a ↵
```

……再び"FUNC-3A.COM" を起動する

```
Ok
```

……こんどは削除できた

```
A>dir new_dir
 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

ファイルが見つかりません

A>■
```

確かになくなっている

Fig. 7.41　実行結果　39H, 3AH, 3BH

FUNCTION　3CH, 3DH, 3EH, 3FH, 40H

ファンクション 3CH　(Create a File)——ファイルの作成

コール手順

AH　　←3CH
DS：DX　←パス名の先頭アドレス
CX　　←ファイル属性※　　※ p. 339　Table 5.1 参照
INT　21H

Return

キャリーがセットされた場合
AX＝03H　指定したパス名が無効であるか，またはパスが存在しない
AX＝04H　オープンされているファイルが多すぎる(オープンできるファイルの数は〝CONFIG．SYS〟ファイルで設定した値によって決まる)
指定された属性のファイルが作成されたが，アクセスのためのファイル・ハンドルに余裕がない，または内部システムテーブルに空き領域がない
AX＝05H　アクセスが否定された
CX レジスタで指定された属性に作成不可能なもの(ディレクトリ，ボリュームラベル)がはいっていたか，ファイルを保護する属性が与えられていた

キャリーがセットされない場合
AX←ファイル・ハンドル

解説

DS：DX で指定されたパス名のファイルが存在しなければ，該当するディレクトリに新規のファイルが作成され，CX レジスタで指定した属性が与えられる．また指定されたファイルがすでに存在している場合には，既存のファイルの大きさを 0 にして，CX レジスタの属性を与える．返されたファイル・ハンドルは，読出し／書込みのためにオープンされている．

ファイル・ポインタはファイルの先頭にセットされ(00000000H：4 バイト)ファイルの読出し

または書込みはこのポインタから開始される．

ファンクション 3DH　(Open a File)——ファイルのオープン

コール手順

```
AH      ←3DH
DS：DX  ←パス名の先頭アドレス
AL      ←0  ファイルを読出しのためにオープンする
          1  ファイルを書込みのためにオープンする
          2  ファイルを読出し／書込みのためにオープンする
INT  21H
```

Return

キャリーがセットされた場合

AX＝02H　ファイルが存在しない

AX＝03H　指定されたパス名が無効であるか，またはパスが存在しない

AX＝04H　オープンされているファイルが多すぎる
処理するためのファイル・ハンドルに余裕がない，または内部システムテーブルに空き領域がない

AX＝05H　アクセスが否定された
ディレクトリまたはボリュームラベルをオープンしようとした．あるいは，リードオンリーのファイルを書込み用にオープンしようとした

AX＝0CH　無効なアクセス
AL レジスタで指定された値の範囲が 0～2 の範囲外である

キャリーがセットされない場合

AX←ファイル・ハンドル

解説

ファイルをオープンしてファイル・ハンドルを用意する．DS：DX にはオープンするファイルの ASCIZ 文字列の先頭アドレスを与える．読出し／書込みのポインタはファイルの先頭バイトにセットされ，このファイルのレコードサイズは 1 となる．

書込み用や読出し／書込み用にオープンしてもファイルを新しく作成しない．

ファンクション 3EH　(Close a File Handle)——ファイルのクローズ

コール手順

```
AH　←3EH
BX　←ファイル・ハンドル
INT　21H
```

Return

キャリーがセットされた場合

AX＝06H　無効なファイル・ハンドル

BX レジスタで指定されたファイル・ハンドルは，現在オープンされていない

キャリーがセットされない場合

エラーなし

解説

BX レジスタで指定されたファイル・ハンドルのファイルがクローズされる．ファイルに書込みを行った場合は，更新されたディレクトリ情報や FAT をディスクに書き込む．また，BX レジスタで指定されたファイル・ハンドルのワークエリアが開放され，ほかのファイル・ハンドルで使用可能となる．

ファンクション 3FH　(Read From File／Device)——ファイル／デバイスの読出し

コール手順

```
AH　　←3FH
DS：DX　←入力バッファの先頭アドレス
CX　　←読み込むバイト数
BX　　←ファイル・ハンドル
INT　21H
```

Return

キャリーがセットされた場合

AX＝05H　アクセスは否定された

BX レジスタで指定されたファイル・ハンドルは，読出しが許可されていない

AX＝06H　無効なファイル・ハンドル

BX レジスタで指定されたファイル・ハンドルは現在オープンされていない

キャリーがセットされない場合

AX←読み込まれたバイト数

解説

CXレジスタで指定されたバイト数のデータをBXレジスタで指定したファイル・ハンドルで扱うファイルから入力バッファ内に転送するが，指定されたバイト数のデータをすべて読み込む保証はない．たとえばキーボード入力の場合，最大1行分のデータしか読み込まれない．また，AXレジスタが0のとき，ファイルの終わりに達していながらさらにデータを読み込もうとしていることを示す．

またファイルの終わりまでのバイト数より大きな値をCXレジスタに指定した場合には，ファイルの終わりまでを読み込んでそのバイト数をAXレジスタに返す．

すべてのデータの入出力は，標準ポインタ※を使用して行われるので，バッファの先頭アドレスから読み込むバイト数がセグメントの範囲を超えた場合でも，連続した領域に正しく読み込まれる．

ファンクション40H　(Write to File／Device)——ファイル／デバイスへの書込み

コール手順

```
AH      ←40H
DS：DX  ←入力バッファの先頭アドレス
CX      ←書き込むバイト数
BX      ←ファイル・ハンドル
INT  21H
```

Return

キャリーがセットされた場合

AX＝05H　アクセスは否定された
　　ファイル・ハンドルは書込みを許可するモードではない

AX＝06H　無効なファイル・ハンドル
　　BXレジスタで指定されたファイル・ハンドルは現在オープンされていない

キャリーがセットされない場合

AX←書き込まれたバイト数

解説

指定されたバイト数のデータを入力バッファからファイルに転送する(書き込む)．書き込まれたバイト数が指定されたバイト数と一致しない場合，何らかのエラーが起こったものと考えられる．

このシステムコールで，CX＝0000H(バイト数＝0)が指定された場合，現在のファイル・ポイン

※標準ポインタ……システム内で用いられる20ビットのポインタ

タの位置でファイルが切り詰められる．

すべての入出力は，標準ポインタを使用して行われ，セグメントのラップアラウンドは起こらない(オフセットがFFFFHを超えたとき，その処理は同一セグメントの0000Hに戻らずに正しく行われる)．つまり，バッファの先頭アドレスから書き込むバイト数がセグメントの範囲(64Kバイト)を超えた場合でも，正しく連続した領域が書き込まれる．

また，指定したバイト数を書き込むだけの領域がディスクにない場合，エラーにはならずにAXレジスタには0000Hが返され，ファイルには何も書き込まれない．

FUNCTION　3CH, 3DH, 3EH, 3FH, 40H　　Sample Program

前に作ったCOPYコマンドに相当するプログラムをXENIXコンパチブルなこれらのシステムコールで実現する．ただしファイル名は“Source name?”“Dest name?”とプロンプトを出して入力を求めるようにする．

```
                        ;-------------------------------------------;
                        ;   Function  3ch: Create a File            ;
                        ;             3dh: Open a File              ;
                        ;             3eh: Close a File Handle      ;
                        ;             3fh: Read From File/Device    ;
                        ;             40h: Write to a File/Device   ;
                        ;-------------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h
                        ;
0100  B4 09             start:  mov     ah,9
0102  BA 0219 R                   mov     dx,offset sname
0105  CD 21                      int     21h
0107  B4 0A                      mov     ah,0ah
0109  BA 023B R                   mov     dx,offset source
010C  CD 21                      int     21h
010E  BE 023D R                   mov     si,offset source[2]
0111  8A 1E 023C R                mov     bl,source[1]
0115  32 FF                      xor     bh,bh
0117  C6 00 00                   mov     byte ptr [bx][si],0

011A  B4 09                      mov     ah,9
011C  BA 022A R                   mov     dx,offset dname
011F  CD 21                      int     21h
0121  B4 0A                      mov     ah,0ah
0123  BA 027B R                   mov     dx,offset dest
0126  CD 21                      int     21h
0128  BF 027D R                   mov     di,offset dest[2]
012B  8A 1E 027C R                mov     bl,dest[1]
012F  32 FF                      xor     bh,bh
0131  C6 01 00                   mov     byte ptr [bx][di],0

0134  B4 3D                      mov     ah,3dh
0136  8B D6                      mov     dx,si
0138  B0 00                      mov     al,0
013A  CD 21                      int     21h
013C  72 37                      jc      openerr
013E  A3 02BB R                   mov     handle_s,ax
```

“Source name ?”と表示し，ソースファイルの入力を求める

入力されたファイル名の最後に00Hを入れ，ASCIZ文字列とする

“Dest name ?”と表示し，ディスティネーションファイルの入力を求める

入力されたファイル名の最後に00Hを入れ，ASCIZ文字列とする

“ファイルのオープン”のシステムコール
ソースファイルを読出しモードでオープンする
エラーならば“openerr”へジャンプする．AXレジスタに返されたファイル・ハンドルを保存しておく

```
0141  B4 3C                          mov     ah,3ch          ; "ファイルの作成"のシステムコール
0143  8B D7                          mov     dx,di           ; ディスティネーションファイルを作成する
0145  B9 0000                        mov     cx,0            ; このとき読出し／書込みモードでオープンされる
0148  CD 21                          int     21h             ; 作成できなければ"makeerr" へジャンプする
014A  72 2F                          jc      makeerr
014C  A3 02BD R                      mov     handle_d,ax

014F  B4 3F                  nxtrec: mov     ah,3fh          ; "ファイルデバイスの読出し"のシステムコール
0151  BA 02BF R                      mov     dx,offset buffer ; 1024バイトを "buffer"に読み込む
0154  B9 0400                        mov     cx,1024
0157  8B 1E 02BB R                   mov     bx,handle_s
015B  CD 21                          int     21h
015D  72 22                          jc      readerr
015F  3D 0000                        cmp     ax,0            ; 読み込まれたバイト数が0なら "cpyend"へジャンプする
0162  74 29                          je      cpyend

0164  8B C8                          mov     cx,ax           ; 読み出したバイト数だけ書き込む
0166  B4 40                          mov     ah,40h          ; エラーならば "wrterr"へジャンプする
0168  BA 02BF R                      mov     dx,offset buffer
016B  8B 1E 02BD R                   mov     bx,handle_d
016F  CD 21                          int     21h
0171  72 14                          jc      wrterr
0173  EB DA                          jmp     nxtrec

0175  BA 01A6 R              openerr:mov     dx,offset msgser ; ソースファイルがない場合のエラーメッセージ
0178  EB 26 90                       jmp     msgout

017B  BA 01C1 R              makeerr:mov     dx,offset msgmer ; 新規のファイルが作成できない場合のエラーメッセージ
017E  EB 20 90                       jmp     msgout

0181  BA 01D5 R              readerr:mov     dx,offset msgrer ; 読込みが正常に行われない場合のエラーメッセージ
0184  EB 1A 90                       jmp     msgout

0187  BA 01E5 R              wrterr: mov     dx,offset msgwer ; 書込みが正常に行われない場合のエラーメッセージ
018A  EB 14 90                       jmp     msgout

018D  B4 3E                  cpyend: mov     ah,3eh          ; ディスティネーションファイルをクローズする. クローズできなければ"clserr" へジャンプする
018F  8B 1E 02BD R                   mov     bx,handle_d
0193  CD 21                          int     21h
0195  72 06                          jc      clserr
0197  BA 0206 R                      mov     dx,offset msgend ; コピー終了のメッセージ
019A  EB 04 90                       jmp     msgout

019D  BA 01F4 R              clserr: mov     dx,offset msgcer ; クローズができない場合のエラーメッセージ

01A0  B4 09                  msgout: mov     ah,9            ; メッセージの出力
01A2  CD 21                          int     21h

01A4  CD 20                          int     20h             ; プログラムの終了
                             ;
01A6  0D 0A 0A 53 6F 75      msgser  db  0dh,0ah,0ah,'Source File Not Found'
      72 63 65 20 46 69              ,0dh,0ah,'$'
      6C 65 20 4E 6F 74
      20 46 6F 75 6E 64
      0D 0A 24
01C1  0D 0A 0A 44 69 72      msgmer  db  0dh,0ah,0ah,'Directory Full',0dh,0ah,'$'
      65 63 74 6F 72 79
      20 46 75 6C 6C 0D
      0A 24
01D5  0D 0A 0A 52 65 61      msgrer  db  0dh,0ah,0ah,'Read error',0dh,0ah,'$'
      64 20 65 72 72 6F
      72 0D 0A 24
01E5  0D 0A 0A 44 69 73      msgwer  db  0dh,0ah,0ah,'Disk Full',0dh,0ah,'$'
      6B 20 46 75 6C 6C
      0D 0A 24
01F4  0D 0A 0A 43 61 6E      msgcer  db  0dh,0ah,0ah,'Cannot Close',0dh,0ah,
                                     '$'
```

```
      6E 6F 74 20 43 6C
      6F 73 65 0D 0A 24
0206  0D 0A 0A 43 6F 70          msgend  db  0dh,0ah,0ah,'Copy Complete',0dh,0ah
                                 ,'$'
      79 20 43 6F 6D 70
      6C 65 74 65 0D 0A
      24
                                 ;
0219  0D 0A 53 6F 75 72          sname   db  0dh,0ah,'Source name ? $'
      63 65 20 6E 61 6D
      65 20 3F 20 24
022A  0D 0A 20 20 44 65          dname   db  0dh,0ah,'  Dest name ? $'
      73 74 20 6E 61 6D
      65 20 3F 20 24
023B  3E 3F [                    source  db      62,63 dup (?)
            ??
                ]

027B  3E 3F [                    dest    db      62,63 dup (?)
            ??
                ]

02BB  ????                       handle_s        dw      ?
02BD  ????                       handle_d        dw      ?
02BF  0400 [                     buffer  db      1024 dup (?)
            ??
                ]

                                 ;
06BF                             cseg    ends
                                 ;
                                         end     start
```

Fig. 7.42　Sample List　3CH, 3DH, 3EH, 3FH, 40H

実行手順　　FUNC－3C↵

```
A>dir subdir1↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1
                ・すべてディレクトリ
.             <DIR>      84-03-24  12:32
..            <DIR>      84-03-24  12:32
SUBDIR2       <DIR>      84-03-24  12:32
       3 個のファイルがあります
   240640 バイトが使用可能です

A>dir subdir1¥subdir2↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1¥subdir2

.             <DIR>      84-03-24  12:32
..            <DIR>      84-03-24  12:32
FUNC-3C  ASM      1987  84-03-07  10:37
       3 個のファイルがあります
   240640 バイトが使用可能です

A>func-3c↵ ……"FUNC-3C.COM" を実行する

Source name ? subdir1¥subdir2¥func-3c.asm ……ソースファイルのパス名を入力する
  Dest name ? subdir1¥3ch.asm ……ディスティネーションファイルのパス名を入力する

Copy Complete ……コピーが完了したメッセージ
```

サブ・ディレクトリ "¥SUBDIR1" にはファイルがなく "¥SUBDIR1 ¥SUBDIR2"には"FUNC-3C.ASM" があることを確認しておく

```
A>dir subdir1

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1

.             <DIR>      84-03-24  12:32
..            <DIR>      84-03-24  12:32
SUBDIR2       <DIR>      84-03-24  12:32
3CH      ASM      1987  84-03-24  16:13
        4 個のファイルがあります
   238592 バイトが使用可能です

A>dir subdir1¥subdir2

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1¥subdir2

.             <DIR>      84-03-24  12:32
..            <DIR>      84-03-24  12:32
FUNC-3C  ASM      1987  84-03-07  10:37
        3 個のファイルがあります
   238592 バイトが使用可能です

A>fc /b subdir1¥3ch.asm subdir1¥subdir2¥func-3c.asm ……FCでファイルの内容を比較する

A>■…… 何のメッセージも出力されずファイルの内容が同一であることがわかる
```

新たに“3CH．ASM”がディレクトリ“¥SUBDIR1”に作成されていることを確認する
このときファイルの大きさは等しいがCOPYコマンドと違って，ファイルの日付がコピーを行った日付になっていることに注意（COPYコマンドではファンクション56Hを用いて日付を保存する）

Fig. 7.43　実行結果　3CH，3DH，3EH，3FH，40H

FUNCTION　41H

ファンクション41H　(Delete Directory Entry)——ファイルの削除

コール手順

AH　　←41H
DS：DX　←パス名の先頭アドレス
INT　21H

Return

キャリーがセットされた場合
AX＝02H　ファイルが存在しない
AX＝03H　指定したパス名が無効であるか，またはパスが存在しない
AX＝05H　アクセスが否定された
　　指定されたパスがディレクトリ，またはリードオンリーのファイルである

キャリーがセットされない場合
エラーなし

解説

指定されたファイルを削除する．このファイルが現在オープンされている場合，削除は行われ

ない．ワイルドカードを用いるとAX＝03Hのエラーリターンを返す．

FUNCTION　41H　　Sample Program

ファイルを削除するDELまたはERASEコマンドに相当するプログラムを作成する．

```
                                ;-----------------------------------------;
                                ;   Function  41h: Delete Directory Entry ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0080                                    org     80h
                                ;
0080    80 [                    cmdln   db      128 dup (?)
            ??
               ]

                                ;
0100                                    org     100h
                                ;
0100  8A 1E 0080 R              start:  mov     bl,cmdln[0]          ┐ パラメータが何もなければ"error"へジャンプする
0104  84 DB                             test    bl,bl                │
0106  74 23                             je      error                ┘
0108  32 FF                             xor     bh,bh                ┐ パラメータの最後(CR)を00Hに変えASCIZ文字列とする
010A  C6 87 0081 R 00                   mov     cmdln[1][bx],0       ┘
010F  B4 41                             mov     ah,41h               ┐ "ファイルの削除"のシステムコール
0111  BA 0082 R                         mov     dx,offset cmdln[2]   │ パラメータで指定されたファイルを削除する
0114  CD 21                             int     21h                  ┘
0116  BA 0134 R                         mov     dx,offset msgok      ┐
0119  73 13                             jnc     msgout               │
011B  BA 013B R                         mov     dx,offset msgnf      │
011E  3D 0002                           cmp     ax,2                 │
0121  74 0B                             je      msgout               │ ファイルが削除できたら"Ok"と表示する
0123  BA 0164 R                         mov     dx,offset msgro      │ エラーで終了したならエラー番号に対応するメッセージを出力する
0126  3D 0005                           cmp     ax,5                 │
0129  74 03                             je      msgout               │
012B  BA 014E R                 error:  mov     dx,offset msgips     │
                                        ;                            │
012E  B4 09                     msgout: mov     ah,9                 │
0130  CD 21                             int     21h                  ┘
                                        ;
0132  CD 20                             int     20h ……プログラムの終了
                                ;
0134  0D 0A 4F 6B 0D 0A         msgok   db  0dh,0ah,'Ok',0dh,0ah,'$'
      24
013B  0D 0A 46 69 6C 65         msgnf   db  0dh,0ah,'File not found',0dh,0ah,'$
'
      20 6E 6F 74 20 66
      6F 75 6E 64 0D 0A
      24
014E  0D 0A 49 6E 76 61         msgips  db  0dh,0ah,'Invalid Path name',0dh,0ah
,'$'
      6C 69 64 20 50 61
      74 68 20 6E 61 6D
      65 0D 0A 24
0164  0D 0A 46 69 6C 65         msgro   db  0dh,0ah,'File Read Only',0dh,0ah,'$
'
      20 52 65 61 64 20
      4F 6E 6C 79 0D 0A
      24
                                ;
0177                            cseg    ends
                                ;
                                        end     start
```

Fig. 7.44　Sample List　41H

実行手順　　FUNC－41 〈パス名〉 ↵

```
A>dir subdir1¥subdir2↵ ……"FUNC-41．ASM" があることを確認する

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1¥subdir2

.            <DIR>      84-03-24  12:32
..           <DIR>      84-03-24  12:32
FUNC-41  ASM       897  84-03-06  18:11
       3 個のファイルがあります
  260096 バイトが使用可能です

A>func-41 subdir1¥subdir2¥func-41.asm↵ ……"FUNC-41．COM" を実行する

Ok ……正常に終了したメッセージ

A>dir subdir1¥subdir2↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1¥subdir2

.            <DIR>      84-03-24  12:32
..           <DIR>      84-03-24  12:32
       2 個のファイルがあります
  261120 バイトが使用可能です

A>■
```

"FUNC-41．ASM" が削除されている

Fig. 7.45　実行結果　41H

FUNCTION　2FH, 42H

ファンクション 2FH　(Get Disk Transfer Address)
——ディスク転送アドレスの読出し

コール手順

```
AH　←2FH
INT　21H
```

Return

ES：BX← DTA

解説

DTA を ES：BX に返す．MS-DOS のデフォルト DTA は，PSP 内のオフセット 80H である．

ファンクション 42H　(Move a File Pointer)——ファイル・ポインタの移動

コール手順

```
AH      ← 42H
CX：DX  ←移動するバイト数(CX＊10000H＋DX)
AL      ←移動方式
          00H：ファイル・ポインタは，ファイルの先頭からオフセットの位置に移動する
          01H：ファイル・ポインタは，現在の位置にオフセットを加算した位置に移動する
          02H：ファイル・ポインタは，ファイルの終わりにオフセットを加算した位置に
               移動する
BX      ←ファイル・ハンドル
INT  21H
```

Return

キャリーがセットされた場合

AX＝01H　無効なファンクション
　　AL レジスタで指定された移動方式の番号が 00H～02H の範囲外である

AX＝06H　無効なファイル・ハンドル
　　BX レジスタで指定されたファイル・ハンドルは，現在オープンされていない

キャリーがセットされない場合

DX：AX ←新規のファイル・ポインタのファイルの先頭からのオフセット値

解説

ファイル・ポインタをALレジスタで指定した移動方式で移動する．オフセットは上位16ビットがCXレジスタ，下位16ビットをDXレジスタによって指定された32ビットの整数とみなされる．

たとえばファイル・ポインタは，00000000H～FFFFFFFFH の任意の位置に移動できるが，ディスクの空き領域より大きな値を指定しておいて，ファンクション 40H を用いて書込みを行おうとしても，書き込んだバイト数に 0 が返されて，実際には何も変更されない．

FUNCTION　2FH, 42H　　Sample Program

ファイルにそのファイル自身を付け加えるプログラムを作成する．ファンクション 2FH でファイルの大きさを求めて，そのバイト数(最大 32768)だけファイルを読み出し，ファイル・ポインタを移動させて，メモリに読み込んだデータをファイルに書き込む．扱うファイルはアスキーファイル(バイナリだと最後に 1 バイトが抜ける)で，ファイルの最後には CTRL-Z(1AH)が 1 個だけはいっているものとする．

```
                        ;-----------------------------------------;
                        ;   Function  2fh: Get Disk Transfer Address  ;
                        ;             42h: Move a File Pointer      ;
                        ;-----------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h
                        ;
0100  B4 09             start:  mov     ah,9                    ┐
0102  BA 01A5 R                 mov     dx,offset prompt        │
0105  CD 21                     int     21h                     │ "Path name ?"と表示しパス名の入力を行う
0107  B4 0A                     mov     ah,0ah                  │
0109  BA 01B4 R                 mov     dx,offset ptname        │ 入力されたパス名の最後を00Hとし，ASCIZ文字列とする
010C  CD 21                     int     21h                     │
010E  BE 01B6 R                 mov     si,offset ptname[2]     │
0111  8A 1E 01B5 R              mov     bl,ptname[1]            │
0115  32 FF                     xor     bh,bh                   │
0117  C6 00 00                  mov     byte ptr [bx][si],0     ┘

011A  B4 4E                     mov     ah,4eh                  ┐
011C  8B D6                     mov     dx,si                   │ "一致するファイルの検索"のシステムコール
011E  B9 0000                   mov     cx,0                    │ DTAにディレクトリ情報が返される
0121  CD 21                     int     21h                     │
0123  72 63                     jc      openerr                 ┘

0125  B4 2F                     mov     ah,2fh                  ┐
0127  CD 21                     int     21h                     │ "ディスク転送アドレスの読出し" のシステムコール
0129  26: 8B 47 1A              mov     ax,es:[1ah][bx]         │ DTAがどこかを知り，DTA+1AHにはいるファイルの大きさ(バイト長)を得る
012D  A3 01F4 R                 mov     fl_size,ax              │
0130  85 C0                     test    ax,ax                   │ ファイルの大きさが0なら "return" へジャンプする
0132  74 4A                     je      return                  ┘

0134  B4 3D                     mov     ah,3dh                  ┐
0136  BA 01B6 R                 mov     dx,offset ptname[2]     │ ファイルを読出し／書込みでオープンする
0139  B0 02                     mov     al,2                    │ オープンできなければ "openerr" へジャンプする
013B  CD 21                     int     21h                     │
013D  72 49                     jc      openerr                 │
013F  A3 01F6 R                 mov     handle,ax               ┘

0142  B4 42                     mov     ah,42h                  ┐
0144  33 C9                     xor     cx,cx                   │
0146  33 D2                     xor     dx,dx                   │ ファイル・ポインタをファイルの先頭に移動する
0148  B0 00                     mov     al,0                    │
014A  8B 1E 01F6 R              mov     bx,handle               │
014E  CD 21                     int     21h                     ┘

0150  B4 3F                     mov     ah,3fh                  ┐
0152  BA 01F8 R                 mov     dx,offset buffer        │
0155  8B 0E 01F4 R              mov     cx,fl_size              │ "buffer" 内にファイル全体を読み込む
0159  8B 1E 01F6 R              mov     bx,handle               │
015D  CD 21                     int     21h                     ┘

015F  B4 42                     mov     ah,42h                  ┐
0161  B9 FFFF                   mov     cx,0ffffh               │
0164  BA FFFF                   mov     dx,0ffffh               │ ファイル・ポインタを(ファイルの最後)－1に移動する
0167  B0 02                     mov     al,2                    │
0169  8B 1E 01F6 R              mov     bx,handle               │
016D  CD 21                     int     21h                     ┘

016F  B4 40                     mov     ah,40h                  ┐
0171  BA 01F8 R                 mov     dx,offset buffer        │ 読み込んだファイルの内容をファイル・ポインタから書込む
0174  8B 0E 01F4 R              mov     cx,fl_size              │
0178  8B 1E 01F6 R              mov     bx,handle               │
017C  CD 21                     int     21h                     ┘
```

```
017E  B4 3E                     return: mov     ah,3eh          ┐
0180  8B 1E 01F6 R                      mov     bx,handle       ├ ファイルをクローズする
0184  CD 21                             int     21h             ┘
0186  CD 20                             int     20h ……プログラムを終了する

0188  B4 09                     openerr:mov     ah,9            ┐
018A  BA 0191 R                         mov     dx,offset msgoer├ エラーメッセージを表示してプログラムを終了する
018D  CD 21                             int     21h             │
018F  CD 20                             int     20h             ┘
                                ;
0191  0D 0A 0A 46 69 6C         msgoer  db  0dh,0ah,0ah,'File Not Found',0dh,0a
                                h,'$'
      65 20 4E 6F 74 20
      46 6F 75 6E 64 0D
      0A 24
01A5  0D 0A 50 61 74 68         prompt  db  0dh,0ah,'Path name ? $'
      20 6E 61 6D 65 20
      3F 20 24
01B4  3E 3F [                   ptname  db      62,63 dup (?)
             ??
                ]

01F4  ????                      fl_size dw      ?
01F6  ????                      handle  dw      ?
01F8  8000 [                    buffer  db      32768 dup (?)
             ??
                ]

                                ;
81F8                            cseg    ends
                                ;
                                        end     start
```

Fig. 7.46　Sample List　2FH, 42H

実行手順　　FUNC−2F 〈パス名〉 ↵

```
A>dir func-2f.asm ↵ ……もとのファイルの大きさを確認する

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥
                    ┌1311バイトになっている
FUNC-2F  ASM      1311  84-03-24  15:48
       1 個のファイルがあります
   221184 バイトが使用可能です

A>type func-2f.asm ↵ ……TYPEコマンドでその内容も確認しておく
;-------------------------------------------;
;   Function  2fh: Get Disk Transfer Address  ;
;             42h: Move a File Pointer        ;
;-------------------------------------------;
;
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     100h
;
start:  mov     ah,9
        mov     dx,offset prompt
        int     21h
;
fl_size dw      ?
handle  dw      ?
buffer  db      32768 dup (?)
;
cseg    ends
;
        end     start
```

```
A>func-2f ……"FUNC-2F.COM" の起動

Path name ? func-2f.asm ……"FUNC-2F.ASM" の後ろに同じものを付け加えたファイルを作り
A>dir func-2f.asm          それを"FUNC-2F.ASM" とする

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

FUNC-2F  ASM      2621  84-03-24  15:49
       1 個のファイルがあります
  220160 バイトが使用可能です                ファイルサイズが(もとのファイルサイズの2倍) - 1になっている

A>type func-2f.asm ……TYPEコマンドで表示させて確認する
;-----------------------------------------------;
;   Function  2fh: Get Disk Transfer Address    ;
;             42h: Move a File Pointer          ;
;-----------------------------------------------;
;
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     100h
;
start:  mov     ah,9
        mov     dx,offset prompt
        int     21h
                0ah
buffer  db      3z                  ame
;
cseg    ends
;
        end     start                                   ここでファイルが結合されている
;-----------------------------------------------;
;   Function  2fh: Get Disk Transfer Address    ;
;             42h: Move a File Pointer          ;
;-----------------------------------------------;
;
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     100h
;
        mov     ah,9
                offset prompt
;
cseg    ends
;
        end     start
A>■
```

Fig. 7.47　実行結果　2FH, 42H

FUNCTION　43H

ファンクション43H　(Change Attributes)——ファイルの属性の変更

コール手順

ファイル属性を得る場合

AH　　←43H
DS：DX　←パス名の先頭アドレス
AL　　←00H
INT　21H

ファイル属性をセットする場合

AH　　←43H
DS：DX　←パス名の先頭アドレス
AL　　←01H
CX　　←ファイル属性
INT　21H

Return

キャリーがセットされた場合

AX＝01H　無効なファンクション　ALレジスタの値が00H～01Hの範囲外である
AX＝02H　該当するファイルが存在しない　パス名にボリュームラベルを指定している
AX＝03H　パス名が無効であるか，またはパスが存在しない
AX＝05H　アクセスが否定された
　　CXレジスタで指定されたファイル属性に，変更できないものが含まれていた(ディレクトリ，ボリュームラベル)

キャリーがセットされない場合

エラーなし(ファイル属性をセットする場合)
CX←ファイルの属性(ファイル属性を得る場合)

解説

パス名(ASCIZ文字列)で指定されたディレクトリ・エントリの属性がCXレジスタに返されるか，変更される．CXレジスタに返される値と属性の関係はTable 7.3のとおりである．

属性をR／O(リードオンリー)にすれば，DELコマンドで削除したり書込みを行ったりすることができない．また，DIRコマンドでは表示されない〝IO．SYS〟や〝MSDOS．SYS〟のように隠されたファイルにすることもできる．

Table 7．3　ファイル属性

属　性	意　味
01H	読出し専用ファイル(R／O)
02H	不可視属性．通常のディレクトリサーチから除外される
04H	システムファイル
08H	ボリュームラベル
10H	ディレクトリ
20H	保存ビット

FUNCTION 43H Sample Program

指定されたファイルがR／WならばR／O，R／OならばR／Wにするプログラムを作成する．

```
                                ;---------------------------------------;
                                ;    Function  43h:  Change attributes  ;
                                ;---------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0080                                    org     80h
                                ;
0080    80 [                    cmdln   db      128 dup (?)
            ??
               ]

                                ;
0100                                    org     100h
                                ;
0100  8A 1E 0080 R              start:  mov     bl,cmdln[0]     ; コマンドラインにパラメータがなければ，"error" へジャンプする
0104  84 DB                             test    bl,bl           ; パラメータがあれば，ASCIZ文字列に変換する
0106  74 21                             je      error
0108  32 FF                             xor     bh,bh
010A  C6 87 0081 R 00                   mov     cmdln[1][bx],0

010F  B4 43                             mov     ah,43h          ; コマンドラインのパラメータで指定されたファイルを読出しモードでオープンする
0111  B0 00                             mov     al,0            ; オープンできなければ "error" へジャンプする
0113  BA 0082 R                         mov     dx,offset cmdln[2]
0116  CD 21                             int     21h
0118  72 0F                             jc      error

011A  81 F1 0001                        xor     cx,1            ; CXレジスタに返された属性が読出しのみならば読出し／書込み可能に，読出し／書込み可能ならば読出しのみにセットする
011E  B4 43                             mov     ah,43h          ; セットできれば "return" へジャンプする
0120  B0 01                             mov     al,1
0122  BA 0082 R                         mov     dx,offset cmdln[2]
0125  CD 21                             int     21h
0127  73 07                             jnc     return

0129  B4 09                     error:  mov     ah,9            ; 無効なパス名とエラーメッセージを出力する
012B  BA 0132 R                         mov     dx,offset msgerr
012E  CD 21                             int     21h

0130  CD 20                     return: int     20h             ; プログラムの終了
                                ;
0132  0D 0A 49 6E 76 61         msgerr  db  0dh,0ah,'Invalid path name',0dh,0ah
                                ,'$'
      6C 69 64 20 70 61
      74 68 20 6E 61 6D
      65 0D 0A 24

                                ;
0148                            cseg    ends
                                ;
                                        end     start
```

Fig. 7.48 Sample List 43H

実行手順　　FUNC－43 〈パス名〉 ↵

```
A>func-43 func-43.asm ……"FUNC-43.ASM"をリードオンリー(読出しのみ)にする

A>del func-43.asm ……"FUNC-43.ASM"はリードオンリーなのでDELコマンドでは削除できずエラーメッセージが
ファイルが見つかりません                                          出力される

A>dir func-43.asm

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

FUNC-43  ASM        635  84-03-06   18:19
       1 個のファイルがあります
  328704 バイトが使用可能です

                                  実際にはファイルはあるがリードオンリーになっていると
                                  DELコマンドでは "ファイルが見つかりません" と
                                  エラーメッセージが表示される

A>func-43 func-43.asm ……"FUNC-43.ASM"をリードオンリー(読出しのみ可)から
                          リードライト(読出し/書込み)可とする
A>del func-43.asm ……今度はDELコマンドでファイルを削除できる

A>dir func-43.asm

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

ファイルが見つかりません                  "FUNC-43.ASM"は削除された

A>■
```

Fig. 7.49　実行結果　43H

MS-DOSには，ファイルの属性を変更するためのコマンドがありませんので，このシステムコールを使って作ってみるのもよいでしょう．

FUNCTION　44H

ファンクション44H　(I/O Control for Devices)
——デバイスに対するI/Oコントロール

コール手順

AH　　←44H
BX　　←ファイル・ハンドル
BL　　←ドライブ(AL＝04H, 05Hの場合のみ有効. 0：カレント・ドライブ, 1：A, 2：B……)
DS：DX　←データもしくはバッファの先頭アドレス
CX　　←入出力を行うデータのバイト数
AL　　←機能コード
INT　21H

Return

キャリーがセットされた場合
AX＝01H　無効な機能コード

ALレジスタの値が00H～07Hの範囲外である

AX＝05H　アクセスが否定された

AL＝02H～05Hの機能コードを指定したコールを行って，コントロール文字列の受渡しができなかった

AX＝06H　無効なファイル・ハンドル

BXレジスタで指定されたファイル・ハンドルは，現在オープンされていない

AX＝0DH　無効なデータ

DXレジスタで指定されたデバイス情報は無効であった

キャリーがセットされない場合

AL＝02H，03H，04H，05Hの場合

AX←読出しまたは書込みの行われたバイト数

AL＝06H，07Hの場合

AL←FFH　読出しまたは書込みが行える

AL←00H　読出しまたは書込みが行えない

解説

オープンしているファイル・ハンドルに関連したデバイス情報を，セットまたは取得するか，デバイスあるいはデバイス・ドライバへコントロール文字列を転送または取得する．

ALレジスタで渡される機能コードには，以下の種類がある．

AL＝00H	DXレジスタにデバイス情報を返す
AL＝01H	DXレジスタのデバイス情報をセットする
AL＝02H	CXレジスタで指定されたバイト数のコントロール文字列を，デバイスコントロールチャネルからDS：DXで指定されたアドレスに読み込む
AL＝03H	CXレジスタで指定されたバイト数のコントロール文字列を，DS：DXで指定されたアドレスからデバイスコントロールチャネルに書き込む
AL＝04H	BLレジスタで指定したドライブに対して，AL＝02Hと同様の処理を行う
AL＝05H	BLレジスタで指定したドライブに対して，AL＝03Hと同様の処理を行う
AL＝06H	入力ステータスを得る．入力可能な場合，AL＝FFHが返され，準備ができていない場合，AL＝00Hが返される
AL＝07H	出力ステータスを得る．出力可能な場合，AL＝FFHが返され，準備できていない場合，AL＝00Hが返される

AL＝00Hまたは01Hの場合，DXレジスタのビットは以下のように定義されている．なお，デバイス情報をセットする場合，上位バイト(DHレジスタ)は00Hでなければならない．

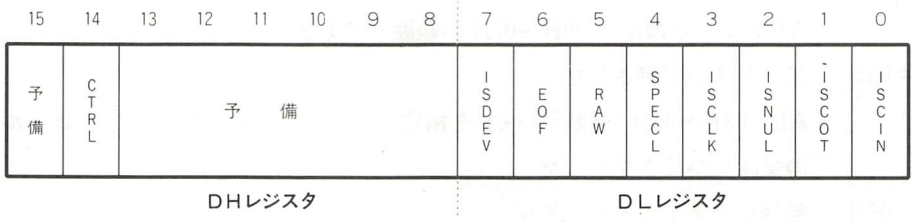


Fig. 7.50　デバイス情報

CTRL＝1　　AL＝02H，AL＝03Hの機能を用いたコールを行って，コントロール文字列の受渡しができる場合

CTRL＝0　　上記ができない場合

ISDEV＝1　　このチャネルがデバイスである場合

　EOF＝0　　EOFを入力する場合

　EOF＝1　　EOFを入力しない場合

　RAW＝0　　このデバイスがCOOKEDモードの場合

　RAW＝1　　このデバイスがRAWモードの場合

　ISCLK＝1　クロックデバイスである場合

　ISNUL＝1　NULデバイスである場合

　ISCOT＝1　コンソール出力デバイスである場合

　ISCIN＝1　コンソール入力デバイスである場合

　SPECL＝1　特殊なデバイス*である場合

ISDEV＝0　　このチャネルがディスクファイルの場合(8～15ビット目は0となる)
0～5ビット目　チャネルと接続したブロック・デバイス番号(00H：A，01H：B……)

AL＝02H～05Hの場合，任意のコントロール文字列をデバイスとの間で転送または取得する．AL＝02H，03Hの場合にデバイス情報のCTRLビットが0であったり，AL＝04H，05Hの場合に無効なドライブが指定されるとAX＝05Hのエラーリターンが返される．

MS-DOSに最初から備わっているデバイスでは，これらの機能は使用できない．ユーザー自身でデバイス・ドライバを書いて使えるようにした場合に初めて有効である．

AL＝06H，07Hの場合，ファイル・ハンドルの入出力の準備ができているかどうか検査できる．

デバイスに対しては，ステータスは前述のとおりだが，ディスクファイルに対してオープンされているファイル・ハンドルのステータスを得る場合，その意味は以下のように定義される．

入力：AL＝FFHの場合　EOFが検出されるまで，つねに準備ができている

※特殊なデバイスとはINT　29Hを用いて，ALレジスタで文字の出力ができることを意味するが，このサポートは将来なくなる可能性がある．

AL＝00Hの場合　ファイル・ポインタを変更しない限り，つねに準備できていない

出力：つねにAL＝FFH

FUNCTION　44H　Sample Program

指定されたファイル名のデバイス(CON，PRNなどを含む)を読出し／書込みでオープンしてそのデバイス情報を取得，表示し，その入力および出力ステータスを表示するプログラムを作成する．

```
                                ;-----------------------------------------;
                                ;   Function  44h: I/O Control for Devices    ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0080                                    org     80h
                                ;
0080    80 [                    cmdln   db      128 dup (?)
            ??
               ]

                                ;
0100                                    org     100h
                                ;
0100  8A 1E 0080 R              start:  mov     bl,cmdln[0]          ┐コマンドラインにパラ
0104  84 DB                             test    bl,bl                │メータがなければerr
0106  74 62                             je      error                │or" へジャンプする
0108  32 FF                             xor     bh,bh                │パラメータがあれば，
010A  C6 87 0081 R 00                   mov     cmdln[1][bx],0       ┘ASCIZ文字列に変換する

010F  B4 3D                             mov     ah,3dh               ┐読出し／書込みモード
0111  BA 0082 R                         mov     dx,offset cmdln[2]   │でファイルをオープン
0114  B0 02                             mov     al,2                 │する
0116  CD 21                             int     21h                  │オープンできないか，
0118  72 50                             jc      error                ┘あるいはファイルがないなら "error" へジャンプする

011A  8B D8                             mov     bx,ax                ┐"デバイスに対するI/
011C  B4 44                             mov     ah,44h               │Oコントロール"のシス
011E  B0 00                             mov     al,0                 │テムコール
0120  CD 21                             int     21h                  │ALレジスタに00Hを
0122  72 46                             jc      error                ┘セットして装置情報を得る

0124  BE 01A2 R                         mov     si,offset devwd      ┐
0127  8A C6                             mov     al,dh                │
0129  E8 0173 R                         call    convert              │
012C  8A C2                             mov     al,dl                │DXレジスタに返され
012E  E8 0173 R                         call    convert              │た装置情報を表示する
0131  B4 09                             mov     ah,9                 │
0133  BA 018A R                         mov     dx,offset msgdev     │
0136  CD 21                             int     21h                  ┘

0138  B4 44                             mov     ah,44h               ┐
013A  B0 06                             mov     al,6                 │ALレジスタに06H
013C  CD 21                             int     21h                  │をセットして入力ステ
013E  72 2A                             jc      error                ┘ータスを得る
```

```
0140  BA 01B8 R                        mov     dx,offset msgrdy
0143  3C FF                           cmp     al,0ffh
0145  74 03                           je      readmsg
0147  BA 01C0 R                        mov     dx,offset msgnrdy
014A  B4 09                    readmsg:mov     ah,9
014C  CD 21                           int     21h

014E  B4 09                           mov     ah,9
0150  BA 01AF R                        mov     dx,offset msgwr
0153  CD 21                           int     21h
0155  B4 44                           mov     ah,44h
0157  B0 07                           mov     al,7
0159  CD 21                           int     21h
015B  72 0D                           jc      error

015D  BA 01B8 R                        mov     dx,offset msgrdy
0160  3C FF                           cmp     al,0ffh
0162  74 09                           je      msgout
0164  BA 01C0 R                        mov     dx,offset msgnrdy
0167  EB 04 90                        jmp     msgout

016A  BA 01CC R                error:  mov     dx,offset msgerr

016D  B4 09                    msgout: mov     ah,9
016F  CD 21                           int     21h

0171  CD 20                           int     20h

0173  50                       convert:push    ax
0174  B1 04                           mov     cl,4
0176  D2 E8                           shr     al,cl
0178  E8 017C R                       call    half
017B  58                               pop     ax
017C  24 0F                    half:   and     al,0fh
017E  04 30                            add     al,30h
0180  3C 3A                           cmp     al,3ah
0182  72 02                           jb      half1
0184  04 07                            add     al,7
0186  88 04                    half1:  mov     [si],al
0188  46                               inc     si
0189  C3                             ret
                               ;
018A  0D 0A 44 65 76 69        msgdev  db  0dh,0ah,'Device Information ..'
      63 65 20 49 6E 66
      6F 72 6D 61 74 69
      6F 6E 20 2E 2E 20
01A2  30 30 30 30 0D 0A        devwd   db      '0000',0dh,0ah,0ah,'Read $'
      0A 52 65 61 64 20
      24
01AF  0D 0A 57 72 69 74        msgwr   db  0dh,0ah,'Write $'
      65 20 24
01B8  52 65 61 64 79 0D        msgrdy  db      'Ready',0dh,0ah,'$'
      0A 24
01C0  6E 6F 74 20 52 65        msgnrdy db      'not Ready',0dh,0ah,'$'
      61 64 79 0D 0A 24
01CC  0D 0A 45 72 72 6F        msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
                               ;
01D6                           cseg    ends
                               ;
                                       end     start
```

読込み可能であれば "Ready" 準備ができていないなら "not Ready" と表示する

メッセージ出力

ALレジスタに07Hをセットして出力ステータスを得る

書込み可能であれば "Ready" 準備ができていないなら "not Ready" と表示する

エラーメッセージ

メッセージの表示

プログラムの終了

[SI], [SI]+1にALレジスタの値を16進2桁の文字列でセットするサブルーチン

Fig. 7.51　Sample List　44H

実行手順　　FUNC－44 〈ファイル名〉 ↵

```
A>func-44 con↵ ……CONに関するデバイス情報を得る
Device Information .. 80D3 ……CONはデバイスでEOFを入力し，COOKEDモードにあるコンソール入出力装置であることがわかる
Read not Ready
Write Ready          } 現在出力はできるが，入力する文字の準備はできていない
A>func-44 aux↵ ……AUXに関するデバイス情報を得る
Device Information .. 80C0 ……AUXはデバイスでEOFを入力しCOOKEDモードになっている
Read not Ready
Write Ready          } 現在出力はできるが，入力する文字の準備はできていない
A>func-44 func-44.asm↵ ……ディスク上のファイル "FUNC-44．ASM" に関するデバイス情報を得る
Device Information .. 0040 ……"FUNC-44．ASM"はディスクファイルでドライブAにあることがわかる
Read Ready
Write Ready          } 現在読出し／書込み可能である
A>func-44 prn↵ ……PRNに関するデバイス情報を得る
Device Information .. 80C0 ……PRNはデバイスでEOFを入力しCOOKEDモードになっている
Read not Ready
Write Ready          } 現在出力可能である
A>■
```

Fig. 7 .52　実行結果　44H

FUNCTION　45H

ファンクション 45H　(Duplicate a File Handle)——ファイル・ハンドルのコピー

コール手順

AH　←45H
BX　←ファイル・ハンドル
INT　21H

Return

キャリーがセットされた場合

AX＝04H　オープンされているファイルが多すぎる
現在のプロセスで使用可能なファイル・ハンドルが存在しなかったか，内部システムテーブルに空き領域がない

AX＝06H　無効なファイル・ハンドル
BX レジスタで指定されたファイル・ハンドルは現在オープンされていない

キャリーがセットされない場合

AX←新規のファイル・ハンドル

解説

BXレジスタで指定された，すでにオープンされているファイル・ハンドルと，同じファイルを扱う新規のファイル・ハンドルを返す．

FUNCTION　45H　　Sample Program

小文字を大文字に変換するフィルタを作成する．

```
                        ;-----------------------------------------------;
                        ;   Function  45h: Duplicate a File Handle     ;
                        ;-----------------------------------------------;
                        ;
0000                    cseg    segment
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h
                        ;
0100  B4 45             start:  mov     ah,45h          ; "ファイル・ハンドルのコピー" のシステムコール
0102  33 DB                     xor     bx,bx           ; 標準入力(ファイル・ハンドル=0) をコピーする
0104  CD 21                     int     21h
0106  8B E8                     mov     bp,ax

0108  B4 3E                     mov     ah,3eh          ; ファイルに割り当てられている標準入力をクローズして，実際のコンソールから入力可能にする
010A  CD 21                     int     21h

010C  B4 45                     mov     ah,45h          ; 標準エラー出力 (ファイル・ハンドル=2) をコピーする
010E  BB 0002                   mov     bx,2
0111  CD 21                     int     21h

0113  B4 3F             aloop:  mov     ah,3fh          ; "ファイル/デバイスからの読出し" のシステムコール
0115  BA 0158 R                 mov     dx,offset buffer ; "buffer"の大きさ128バイトを標準入力をコピーしたファイル・ハンドルから読出す
0118  B9 0080                   mov     cx,128
011B  8B DD                     mov     bx,bp
011D  CD 21                     int     21h
011F  0B C0                     or      ax,ax           ; 読込まれたバイト数が0でないなら"setcx"へジャンプする
0121  75 0A                     jnz     setcx

0123  B4 09                     mov     ah,9            ; ファイルの終わりなら，CR，LFを出力して "return" へジャンプする
0125  BA 0155 R                 mov     dx,offset crlf
0128  CD 21                     int     21h
012A  EB 27 90                  jmp     return

012D  8B C8             setcx:  mov     cx,ax           ; LODSBを使用するための初期設定
012F  8B F2                     mov     si,dx
0131  FC                        cld

0132  AC               tloop:  lodsb                   ; "buffer"内の文字をALレジスタに順に読み込み，英小文字は大文字に変換して出力する
0133  3C 1A                    cmp     al,1ah
0135  74 1C                    jz      return
0137  3C 61                    cmp     al,'a'
0139  72 06                    jb      notlow
013B  3C 7A                    cmp     al,'z'
013D  77 02                    ja      notlow
013F  2C 20                    sub     al,'a'-'A'
0141  B4 02             notlow: mov     ah,2
0143  8A D0                    mov     dl,al
0145  CD 21                    int     21h
```

```
0147  B4 06                          mov     ah,06h
0149  B2 FF                          mov     dl,0ffh
014B  CD 21                          int     21h
014D  75 04                          jnz     return
014F  E2 E1                          loop    tloop

0151  EB C0                          jmp     aloop

0153  CD 20                    return: int     20h
                                     ;
0155  0D 0A 24                 crlf    db      0dh,0ah,'$'
0158      80 [                buffer  db      128 dup (?)
            ??
               ]

                                     ;
0108                           cseg    ends
                                     ;
                                     end     start
```

コンソールからキー入力があれば "return" へジャンプする．キー入力がなければ "tloop" に戻って出力を続ける

aloop ····· "aloop" へ戻って次のデータを読み込む

20h ····· プログラムの終了

Fig. 7.53 Sample List 45H

実行手順　　FUNC−45 〈ファイル名〉 ↵

```
A>type func-45.asm↵ ……"FUNC-45.ASM"には小文字が使われていることを確認する
;-----------------------------------------;
;   Function  45h: Duplicate a File Handle    ;
;-----------------------------------------;
;
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     100h
;
start:  mov     ah,45h
        xor     bx,bx
        int     21h
        mov     bp,ax
buffer  db
;
cseg    ends
;
        end     start

A>func-45 < func-45.asm↵ ……まず "FUNC-45.COM" をI/Oのリダイレクトで実行する
;-----------------------------------------;
;   FUNCTION  45H: DUPLICATE A FILE HANDLE    ;小文字が大文字となって表示される
;-----------------------------------------;
;
CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
;
        ORG     100H
;
START:  MOV     AH,45H
        XOR     BX,BX
        INT     21H
        MOV     BP,AX
                AH,3EH
;
CSEG    ENDS
;
        END     START
```

```
A>type func-45.asm ¦ func-45 ↵……今度はパイプで実行する
;-----------------------------------------;
;   FUNCTION  45H: DUPLICATE A FILE HANDLE    ;
;-----------------------------------------;
;
CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG
;
        ORG     100H
;
START:  MOV     AH,45H
        XOR     BX,BX
        INT     2 ……ここでキー入力があったので出力は中断された
A>■
```

Fig. 7 .54　実行結果　4 5 H

このシステムコールを用いてパイプやリダイレクトを行うと，標準入力は実際にはコンソールにつながっていなくても，コンソールから入力を行うことができます．

また，ディスク上に読出し／書込みのファイルを作り，それをコピーした片方のファイル・ハンドルで，ファイル・ポインタを変えるともう一方でも同じ状態になっています．

FUNCTION　4 6 H

ファンクション 46H　(Force a Duplicate of a Handle)
——指定したファイル・ハンドルへのコピー

コール手順

AH　←46H
BX　←既存のファイル・ハンドル
CX　←新規に作成するファイル・ハンドル
INT　21H

Return

キャリーがセットされた場合

AX＝04H　オープンされているファイルが多すぎる．現在のプロセスで使用可能なファイル・ハンドルが存在しないか，内部システムテーブルに空き領域がない

AX＝06H　無効なファイル・ハンドル．BX レジスタで指定されたファイル・ハンドルは，現在オープンされていない

キャリーがセットされない場合

エラーなし

解説

すでにオープンされているファイル・ハンドルを取り出し，同じファイル・ポインタを持ち，

ファイル名，ステータス，属性といった情報を示す新規のファイル•ハンドルを返す．CX レジスタで指定されたファイル・ハンドルがオープンされている場合，まずそのファイル・ハンドルがクローズされる．

自分でファイル•ハンドルを指定する以外は，ファンクション 45H と同じである．したがって，片方のファイル•ハンドルを用いて読み出した後，別のファイル•ハンドルを用いて読み出すと，前で読んだ次のデータから読み出される．

FUNCTION 46H Sample Program

ファイルをオープンしてそのファイル・ハンドルをコピーし，両方のファイル・ハンドルから交互に文字を読み込んで，標準出力に出力するプログラムを作成する．

```
                                ;-----------------------------------------------;
                                ;   Function  46h: Force a Duplicate            ;
                                ;                              of a Handle      ;
                                ;-----------------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0080                                    org     80h
                                ;
0080    80 [                    cmdln   db      128 dup (?)
            ??
                ]

                                ;
0100                                    org     100h
                                ;
0100  8A 1E 0080 R              start:  mov     bl,cmdln[0]
0104  84 DB                             test    bl,bl
0106  74 3D                             je      error
0108  32 FF                             xor     bh,bh
010A  C6 87 0081 R 00                   mov     cmdln[1][bx],0
010F  B4 3D                             mov     ah,3dh
0111  BA 0082 R                         mov     dx,offset cmdln[2]
0114  B0 00                             mov     al,0
0116  CD 21                             int     21h
0118  72 2B                             jc      error
011A  A3 0180 R                         mov     handle1,ax
011D  8B D8                             mov     bx,ax
011F  40                                inc     ax
0120  A3 0182 R                         mov     handle2,ax
0123  8B C8                             mov     cx,ax
0125  B4 46                             mov     ah,46h
0127  CD 21                             int     21h
0129  72 1A                             jc      error

012B  8B 1E 0180 R              getlp:  mov     bx,handle1
012F  E8 014E R                         call    getchr
0132  72 0B                             jc      endret
0134  8B 1E 0182 R                      mov     bx,handle2
0138  E8 014E R                         call    getchr
013B  72 02                             jc      endret
013D  EB EC                             jmp     getlp

013F  BA 016E R                 endret: mov     dx,offset msgend
0142  EB 04 90                          jmp     msgout
```

コマンドラインにパラメータがなければ"error"へジャンプする
パラメータがあればASCIZ文字列に変換する

コマンドラインで指定されたファイルを読出しモードでオープンする．オープンできなければ"error"へジャンプする

オープンしたとき返されたファイル・ハンドルをファイル・ハンドル＋1の新たなファイル・ハンドルにコピーする

オープンしたファイル・ハンドルとコピーしたファイル・ハンドルから交互に1文字ずつ読み出して，ファイルの終わりまで表示する

終了メッセージ

```
0145  BA 0176 R                 error:  mov     dx,offset msgerr ……エラーメッセージ

0148  B4 09                     msgout: mov     ah,9                  } メッセージの表示
014A  CD 21                             int     21h                   }

014C  CD 20                             int     20h ……プログラムの終了

014E  B4 3F                     getchr: mov     ah,3fh
0150  BA 0184 R                         mov     dx,offset buffer
0153  B9 0001                           mov     cx,1
0156  CD 21                             int     21h
0158  3D 0001                           cmp     ax,1
015B  75 0F                             jne     errret
015D  8A 16 0184 R                      mov     dl,buffer
0161  80 FA 1A                          cmp     dl,1ah
0164  74 06                             je      errret
0166  B4 02                             mov     ah,2
0168  CD 21                             int     21h
016A  F8                                clc
016B  C3                                ret
016C  F9                        errret: stc
016D  C3                                ret
                                ;
016E  0D 0A 45 6E 64 0D         msgend  db      0dh,0ah,'End',0dh,0ah,'$'
      0A 24
0176  0D 0A 45 72 72 6F         msgerr  db      0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
0180  ????                      handle1 dw      ?
0182  ????                      handle2 dw      ?
0184  ??                        buffer  db      ?
                                ;
0185                            cseg    ends
                                ;
                                        end     start
```

BXレジスタで示されるファイル・ハンドルから1文字読み出し，それを表示するサブルーチン　ファイルの終わりであるか，あるいはエラーが起こった場合にはキャリーをセットし，正常に処理が行われた場合にはキャリーをリセットする

Fig. 7.55　Sample List　46H

実行手順　　FUNC−46 [〈ドライブ名：〉]〈パス名〉 ↵

```
A>func-46 func-46.asm ↵……"FUNC-46.COM" を実行する
;-----------------------------------------------;
;   Function  46h: Force a Duplicate            ; ファイル・ハンドルをコピーして両方の
;                          of a Handle          ; ファイル・ハンドルから交互にデータを入
;-----------------------------------------------; 力しているが同じ文字が2度読まれること
;                                                 はない
cseg    segment
        assume  cs:cseg,ds:cseg
;
        org     80h
;
cmdln   db      128 dup (?)
;
        org     100h
;
start:  mov     bl,cmdln[0]
        test    bl,bl
        je      error
        xor     bh,bh
        mov     cmdln[1][bx],0
;
cseg    ends
;
        end     start

End ……終了のメッセージ

A>■
```

Fig. 7.56　実行結果　46H

FUNCTION 47H

ファンクション47H (Return Text of Current Directory)
——カレント・ディレクトリの読出し

コール手順

AH　　←47H
DS：SI　←返される64バイトの情報に対するバッファの先頭アドレス
DL　　←ドライブ番号(0：カレント・ドライブ，1：A，2：B……)
INT　21H

Return

キャリーがセットされた場合

AX＝0FH　ドライブ番号が無効である．このときバッファの内容は変更されない

キャリーがセットされない場合

エラーなし

解説

DS：SIで指定されるアドレスから64バイトの領域に，ディレクトリのルートからの相対的なパス名が，次のようにASCIZ文字列の形で返される．

〈ディレクトリ名〉¥〈ディレクトリ名〉¥……¥〈ディレクトリ名〉00……
　　↑
バッファの残りの部分は変更されない

FUNCTION 47H　Sample Program

システムコールを行い，返された64バイトの内容をダンプ出力するプログラムを作成する．これは"CHDIR↵"からドライブ名の表示を除いたものに相当する．

```
                        ;-----------------------------------------;
                        ;   Function  47h: Return Text of        ;
                        ;                        Current Directory  ;
                        ;-----------------------------------------;
                        ;
0000                    cseg    segment
                                extrn   dump:near,crlfout:near
                                assume  cs:cseg,ds:cseg
                        ;
0100                            org     100h
                        ;
0100  B4 47             start:  mov     ah,47h
0102  BE 011A R                  mov     si,offset buffer
0105  B2 00                     mov     dl,0
0107  CD 21                     int     21h
```

"dump"と"crlf out"を外部サブルーチンとして定義する

"カレント・ディレクトリの読出し"のシステムコールDLレジスタを00Hとし，カレント・ドライブのカレント・ディレクトリを得る

```
0109  E8 0000 E                        call    crlfout
010C  BB 011A R                        mov     bx,offset buffer
010F  B9 0040                          mov     cx,64
0112  E8 0000 E                        call    dump
0115  E8 0000 E                        call    crlfout
0118  CD 20                            int     20h
                                ;
011A      40 [                  buffer  db      64 dup (?)
            ??
               ]
                                ;
015A                            cseg    ends
                                ;
                                        end     start
```

"butter"内の64バイトをダンプ形式で出力する

プログラムの終了

Fig. 7.57　Sample List　47H

実行手順　　FUNC−47 ↵

```
A>chdir ↵
A:¥
```
カレント・ディレクトリはA：¥(ルート・ディレクトリ)である

```
A>func-47 ↵
```
……FUNC-47.COMを実行する

ルートディレクトリなので００Ｈのみが返される

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................

A>chdir sUbdIr1¥subDIR2¥SubDir3 ↵
```
……カレント・ディレクトリを変更する

```
A>dir ↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥sUbdIr1¥subDIR2¥SubDir3
```
……CHDIRコマンドで入力したとおりに表示される

```
.            <DIR>      84-03-24  12:33
..           <DIR>      84-03-24  12:33
       2 個のファイルがあります
  285696 バイトが使用可能です

A>func-47 ↵
```
……再びFUNC-47.COMを実行

DIRコマンドと同じくCHDIRでコマンド入力したとおりにカレント・ディレクトリ名が返される

```
73 55 62 64 49 72 31 5C 73 75 62 44 49 52 32 5C    sUbdIr1¥subDIR2¥
53 75 62 44 69 72 33 00 00 00 00 00 00 00 00 00    SubDir3.........
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................

A>■
```

Fig. 7.58　実行結果　47H

ここで，ディレクトリの名前を取っておくバッファが64バイトしかないということは，このバッファにはいりきらないようなディレクトリは作ることはできても，そのディレクトリには移ることができないということです．すなわち，

63バイト

CHDIR ¥A¥B¥……¥XX ↵

は可能でも，

64バイト

CHDIR ¥A¥B¥……¥XX ↵

は不可能なのです．

FUNCTION 48H, 49H, 4AH

ファンクション48H (Allocate Memory)——メモリブロックの割当て

コール手順

```
AH   ←48H
BX   ←割り当てるメモリ領域のパラグラフサイズ
INT  21H
```

Return

キャリーがセットされた場合

AX＝07H　メモリ・コントロールブロックが破壊されている
メモリ・コントロールブロックに，矛盾したデータが含まれている．これは，ユーザーがメモリ・コントロールブロックを破壊したために起こる

AX＝08H　十分な大きさのメモリがない
使用可能な最大の空きブロックが要求されたものより小さいか，あるいは空きブロックがない

BX ←割当て可能な最大のメモリ領域のパラグラフサイズ

キャリーがセットされない場合

AX ←割り当てられたメモリの先頭のセグメント・アドレス

解説

BX レジスタによって与えられたパラグラフの大きさのメモリを割り当てる．それによってユーザーは，起動時に割り当てられたメモリ以外のメモリを所有できるようになる．

メモリ管理は次の Fig. 7.59 のような形で行われ，このメモリ・コントロールブロックを破壊するとメモリ・アロケーションに関するシステムコールを行った場合に，AX＝07H のエラーリターンが返される．

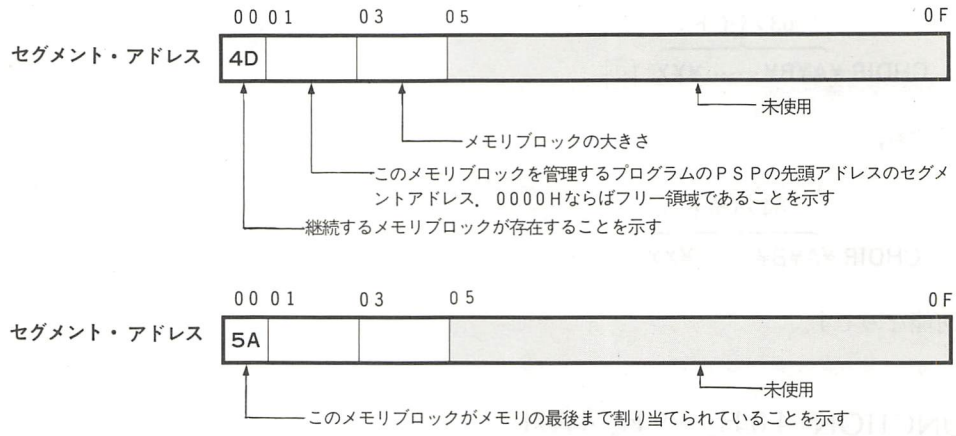


Fig. 7 .59　メモリ・コントロールブロック

一般にユーザープログラムは，メモリの終わりまですでにアロケートされているので，ファンクション 48H を行うためには，まず ES レジスタに PSP のセグメント・アドレスをセットし，ファンクション 49H または 4AH を行ってメモリの空き領域を確保する必要がある．

ファンクション 49H　(Free Allocated Memory)——メモリブロックの解放

コール手順

```
AH　← 49H
ES　←解放するメモリ領域のセグメント・アドレス
INT　21H
```

Return

キャリーがセットされた場合

AX＝07H　メモリ・コントロールブロックが破壊されている
　　メモリ・コントロールブロックに，矛盾したデータが含まれている．これは，ユーザープログラムがメモリ・コントロールブロックを破壊したために起こる

AX＝09H　無効なブロック
　　ES レジスタが示すメモリ領域は，メモリ・アロケーションによって割り当てられたものではない

キャリーがセットされない場合

エラーなし

解説

メモリ・アロケーションによって割り当てられたメモリを，システムの管理下に戻す．

ここでESレジスタは，ファンクション48Hで説明したメモリ・コントロールブロックの次のセグメント・アドレスを指していなければならない．これは，ユーザープログラムにおいてはPSPのセグメント・アドレスである．

ファンクション4AH　(Modify Allocated Memory Blocks)
——メモリブロックのサイズの変更

コール手順

```
AH　←4AH
ES　←メモリ領域のセグメント・アドレス
BX　←割当てを要求するメモリ領域のパラグラフサイズ
INT　21H
```

Return

キャリーがセットされた場合

AX＝07H　メモリ・コントロールブロックが破壊されている
メモリ・コントロールブロックに，矛盾したデータが含まれている．これは，ユーザーがメモリ・コントロールブロックを破壊したために起こる

AX＝08H　十分な大きさのメモリがない
使用可能な最大の空きブロックが，要求されたものより小さいかあるいは空きブロックがない．BXレジスタには割当て可能な最大のメモリ領域のパラグラフサイズ

AX＝09H　無効なブロック
ESレジスタが示すメモリ領域は，メモリ・アロケーションによって割り当てられたものではない

キャリーがセットされない場合

エラーなし

解説

割り当てられたメモリブロックの大きさを拡大，縮小する．

FUNCTION　48H, 49H,　4AH　　Sample Program

メモリ・アロケーションを行うプログラムを作成する．起動時はメモリの最上位までアロケートされているので，まずそれをプログラムの大きさで切り詰め(ファンクション4AH)，その後ろに新しいメモリブロックを作成し(ファンクション48H)，次にそのメモリブロックを解放(ファンクション49H)する．このとき，メモリ・コントロールブロックも表示する．

```
                                ;-------------------------------------------;
                                ;   Function  48h: Allocate Memory          ;
                                ;             49h: Free Allocated Memory    ;
                                ;             4ah: Modify Allocated Memory  ;
                                ;                                  Blocks   ;
                                ;-------------------------------------------;
                                ;
0000                            cseg    segment
                                        extrn   hexout:near,putch:near
                                        extrn   dump:near
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
                                ;
0100  E8 0157 R                 start:  call    alloc

0103  B4 09                             mov     ah,9
0105  BA 0192 R                         mov     dx,offset msg4a
0108  CD 21                             int     21h

010A  B4 4A                             mov     ah,4ah
010C  BB 1000                           mov     bx,1000h
010F  CD 21                             int     21h

0111  E8 0157 R                         call    alloc
0114  8C D8                             mov     ax,ds
0116  05 1001                           add     ax,1000h+1
0119  8E C0                             mov     es,ax
011B  E8 0157 R                         call    alloc

011E  B4 09                             mov     ah,9
0120  BA 0170 R                         mov     dx,offset msg48
0123  CD 21                             int     21h

0125  B4 48                             mov     ah,48h
0127  BB 0010                           mov     bx,10h
012A  CD 21                             int     21h
012C  8E C0                             mov     es,ax

012E  E8 0157 R                         call    alloc
0131  06                                push    es
0132  8C C0                             mov     ax,es
0134  05 0011                           add     ax,10h+1
0137  8E C0                             mov     es,ax
0139  E8 0157 R                         call    alloc
013C  07                                pop     es

013D  B4 09                             mov     ah,9
013F  BA 0181 R                         mov     dx,offset msg49
0142  CD 21                             int     21h

0144  B4 49                             mov     ah,49h
0146  CD 21                             int     21h

0148  E8 0157 R                         call    alloc
014B  8C C0                             mov     ax,es
014D  05 0011                           add     ax,10h+1
0150  8E C0                             mov     es,ax
0152  E8 0157 R                         call    alloc

0155  CD 20                             int     20h
```

"hexout", "putch", "dump" を外部サブルーチンとして定義する

プログラムのPSPの直前のセグメント・アドレスにあるメモリ・コントロールブロックを表示する

"Function 4AH"と表示する

"メモリブロックのサイズの変更" のシステムコール. メモリを64Kバイトで切り詰める

プログラムの前後のメモリ・コントロールブロックを表示する

"Function 48H"と表示する

"メモリのブロックの割当て" のシステムコール
解放したメモリブロックに新たに100Hバイトのメモリブロックを確保する

確保したメモリ領域の前後のメモリ・コントロールブロックを表示する

"Function 49H"と表示する

ファンクション48Hで割当てたメモリブロックを解放する

解放したメモリブロックの前後のメモリ・コントロールブロックを表示する

プログラムの終了

```
0157  1E                        alloc:  push    ds
0158  8C C0                            mov     ax,es
015A  48                              dec     ax
015B  8E D8                           mov     ds,ax
015D  E8 0000 E                       call    hexout
0160  B0 20                           mov     al,' '
0162  E8 0000 E                       call    putch
0165  BB 0000                         mov     bx,0
0168  B9 0010                         mov     cx,10h
016B  E8 0000 E                       call    dump
016E  1F                              pop     ds
016F  C3                              ret

                                      ;
0170  0D 0A 46 75 6E 63       msg48   db  0dh,0ah,'Function 48h',0dh,0ah,'$'
      74 69 6F 6E 20 34
      38 68 0D 0A 24
0181  0D 0A 46 75 6E 63       msg49   db  0dh,0ah,'Function 49h',0dh,0ah,'$'
      74 69 6F 6E 20 34
      39 68 0D 0A 24
0192  0D 0A 46 75 6E 63       msg4a   db  0dh,0ah,'Function 4ah',0dh,0ah,'$'
      74 69 6F 6E 20 34
      61 68 0D 0A 24
                                      ;
01A3                          cseg    ends
                                      ;
                                      end     start
```

(ESレジスタの値－1)で示されるセグメント・アドレスの16バイトをダンプ形式で表示する
メモリ・コントロールブロックは実際は先頭から5バイトまでが有効である

Fig. 7.60 Sample List 48H, 49H, 4AH

実行手順　　FUNC－48↵

```
A>func-48↵ ……"FUNC-48.COM" を起動する
124A 5A 4B 12 B5 2D 20 09 20 09 0D 0A 20 09 20 09 20   ZK.ｵ- . ... . .
                └メモリの上限(124B0H+2DB50H=40000H)までこのメモリブロックはアロケートされている
Function 4ah ┌このメモリブロックをファンクション4AHで10000Hバイトの大きさに切り詰めた
124A 4D 4B 12 00 10 20 09 20 09 0D 0A 20 09 20 09 20   MK... . ... . .
224B 5A 00 00 B4 1D 00 00 00 00 00 00 00 00 00 00 00   Z..I............
          └その後ろがメモリの最後まで割り当てられた空きメモリブロックとなっている
Function 48h ┌空きメモリブロックに100Hバイトの大きさの新たなメモリブロックを作った. これが124B0Hからはじまるプログラムにより割り当てられたものであることがわかる
224B 4D 4B 12 10 00 00 00 00 00 00 00 00 00 00 00 00   MK..............
225C 5A 00 00 A3 1D 00 00 00 00 00 00 00 00 00 00 00   Z..｣............
          └新たな空きメモリブロック
Function 49h ┌このメモリブロックをファンクション49Hで開放したので0000Hが書き込まれている
224B 4D 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00   M...............
225C 5A 00 00 A3 1D 00 00 00 00 00 00 00 00 00 00 00   Z..｣............
          └この部分は前のままである
A>■
```

Fig. 7.61 実行結果 48H, 49H, 4AH

FUNCTION 31H, 4BH, 4CH, 4DH

ファンクション31H　(Keep Process)——プログラムの常駐終了

コール手順

AH　←31H
AL　←リターンコード(子プロセスから親プロセスに受け渡す1バイトのデータ)
DX　←メモリに常駐されるプログラムのパラグラフサイズ
INT　21H

Return

なし

解説

現在のプロセスを常駐させたまま終了する．このとき，DXレジスタで指定したパラグラフサイズのメモリ領域がプログラム領域として確保され，メモリに常駐する．このシステムコールは，ファンクション4CHによるプロセスの終了と違い，プロセスに属するアロケーションを開放しない．ALレジスタに与えたリターンコードは，ファンクション4DHを用いて親プロセスで取得できる．

このシステムコールと同じく，プログラムを常駐させたまま終了させるにはINT　27Hを用いる．これは，MS-DOS ver. 1.25と互換性がある．

ファンクション4BH　(Load and Execute a Program)——プログラムのロードと実行

コール手順

AH　←4BH
DS：DX　←読込みまたは実行させようとするファイルのパス名の先頭アドレス
ES：BX　←パラメータブロックの位置
AL＝00H　プログラムを読み込み，実行する
AL＝03H　プログラムを読み込むのみ
INT　21H

Return

キャリーがセットされた場合
AX＝01H　無効なファンクション
　　ALレジスタの値が正しくない

AX＝02H　ファイルが存在しない

AX＝03H　指定されたパスが存在しないか，あるいは無効である

AX＝08H　十分な大きさのメモリがない

作成すべきプロセスのための十分なメモリ領域が存在しない

AX＝0AH　不正な環境

実行環境が 32K バイト以上であった

AX＝0BH　不正なフォーマット

DS：DX によって指定されたファイルが EXE モデルのファイルであり，そのヘッダ部分に矛盾した情報がはいっていた

キャリーがセットされない場合

エラーなし

解説

あるプログラムから他のプログラムをロードしたり，実行させたりする．プログラムをロード実行させる場合は，メモリの空きブロックにプログラムがロードされパラメータブロックで指定した環境や FCB，コマンドラインの情報がコマンドラインからプログラムを実行した場合と同じく PSP にセットされ制御が移される．これを用いれば，エディタの中からコンパイラやアセンブラを起動するようなアプリケーションなどの開発も容易にできる．

ロードのみ行う場合はオーバーレイ形式でプログラムをロードするのに有効である．

AL レジスタの値により，パラメータブロックは Fig. 7.62，63 のように構成される．

AL ＝ 00H の場合

+00H	1ワード	環境のセグメント・アドレス
+02H	2ワード	PSPのオフセット80Hに作成されるコマンド行に与える80Hバイトのデータのポインタ
+06H	2ワード	PSPのオフセット5CHに作成されるデフォルトFCBに与える10Hバイトのデータのポインタ
+0AH	2ワード	PSPのオフセット6CHに作成されるデフォルトFCBに与える10Hバイトのデータのポインタ

Fig. 7.62　パラメータブロック①

AL ＝ 03H の場合

+00H	1ワード	ファイルをロードするセグメント・アドレス
+02H	1ワード	イメージに対して使用される再配置因子

Fig. 7.63　パラメータブロック②

再配置因子は，EXEのファイルのリロケータブルなセグメント情報を，実際にプログラムで使用できる絶対アドレスに変更するために用いられる．

親プロセス中のオープンされたすべてのファイルは，子プロセスが実行されるとコピーされる．たとえば，親プロセスで標準入力のファイルと標準出力のリスティングファイルをオープンしておいて，標準入出力を用いるソート・プログラムを実行し，ファイルに書き込むことができる．

またプロセス実行環境として，セグメント・アドレスで指定された環境が親プロセスから子プロセスに渡される．環境のフォーマットはFig. 7.64のとおりである．

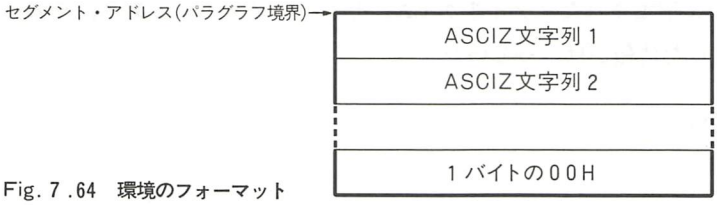


Fig. 7.64　環境のフォーマット

この環境は，32Kバイト未満でなければならない．

1つのASCIZ文字列で示される環境変数と文字列は，以下のフォーマットで構成される．

環境変数＝文字列00H

環境のセグメント・アドレスが0000Hの場合，子プロセスは親プロセスの環境をそのまま受け継ぐ．また，"COMMAND．COM"がプロセスを起動するときに渡す環境はSETコマンドで表示されるものに一致する．

ファンクション4CH　(Terminate a Process)——プロセスの終了

コール手順

```
AH　←4CH
AL　←リターンコード
INT　21H
```

Return

なし

解説

現在のプロセスを終了し，親プロセスに制御を移行する．さらにリターンコードを転送することができる．このとき，オープンされているファイルがクローズされる．この方法は，INT　20HやJMP 0でプロセスを終了する場合に比べてCSレジスタがPSPを指していない場合でも使用できるので便利である．

ファンクション 4DH (Retrieve the Return Code of a Child)
——子プロセスの終了コードの読出し

コール手順

```
AH  ←4DH
INT  21H
```

Return

AH＝00H　終了(ファンクション 00H, 4CH, INT　20H)
AH＝01H　CTRL-C による終了(INT　23H)
AH＝02H　致命的エラー(INT　24H)
AH＝03H　常駐したまま終了(ファンクション 31H, INT　27H)
AL←リターンコード

解説

子プロセスを起動した親プロセスが，子プロセスがどのような条件のもとで終了したかを知るために用いる．プロセスの終了には，ファンクション 00H, 4CH, 内部割込み 20H などで正常に終了する場合，CTRL-C による場合，致命的エラーによる場合，ファンクション 31H, 内部割込み 27H により常駐したまま終了する場合があるが，子プロセスがどの状態で終了したかを親プロセスが AH レジスタの値として受け取ることがこのシステムコールにより可能となる．また ver. 2.0 になって追加されたシステムコール，ファンクション 31H および 4CH で終了時に MS-DOS に渡すリターンコードも AL レジスタにともに受け取ることができる．従来の終了方法ではこのリターンコードは 00H であるが，これを利用して親プロセスでそれぞれの場合に対応する処理を行うようにできる(例：バッチコマンドの ERRORLEVEL)．ただし AX レジスタに返される終了状態コードは一度だけこのシステムコールで得られる．再びこのシステムコールを行っても AX レジスタには 0000H が返される．

FUNCTION　31H, 4BH, 4CH, 4DH　Sample Program

親プロセスから子プロセスを起動して子プロセスからリターンコードを受け取り，表示するプログラムを作成する．このとき子プロセスでは，親プロセスから得た PSP の内容を表示させ，パラメータの受渡しがなされていることを確認する．また，これにより任意のプログラムを起動させることも可能である．

```
                                ;---------------------------------------;
                                ;    Function  31h:  Keep Process       ;
                                ;---------------------------------------;
                                ;
0000                            cseg    segment
                                        extrn   crlfout:near,hexout:near
                                        extrn   dump:near
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
                                ;
0100  E8 0000 E                 start:  call    crlfout
0103  8C D8                             mov     ax,ds
0105  E8 0000 E                         call    hexout
0108  E8 0000 E                         call    crlfout

010B  BB 0000                           mov     bx,0
010E  B9 0100                           mov     cx,100h
0111  E8 0000 E                         call    dump

0114  B4 31                             mov     ah,31h
0116  B0 31                             mov     al,31h
0118  BA 1000                           mov     dx,1000h
011B  CD 21                             int     21h
                                ;
011D                            cseg    ends
                                ;
                                        end     start
```

"crlfout","hexout","dump" を外部サブルーチンとして定義する

PSPのセグメントアドレスを表示する

PSPの先頭から100Hバイトを表示する

リターンコード4CHを転送し，ブロックサイズをスタックを含めたCOMファイルの大きさ，10000Hバイトに設定しプロセスを常駐させて終了する

```
                                ;---------------------------------------;
                                ;   Function  4ch: Terminate a Process  ;
                                ;---------------------------------------;
                                ;
0000                            cseg    segment
                                        extrn   crlfout:near,hexout:near
                                        extrn   dump:near
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
                                ;
0100  E8 0000 E                 start:  call    crlfout
0103  8C D8                             mov     ax,ds
0105  E8 0000 E                         call    hexout
0108  E8 0000 E                         call    crlfout

010B  BB 0000                           mov     bx,0
010E  B9 0100                           mov     cx,100h
0111  E8 0000 E                         call    dump

0114  B4 4C                             mov     ah,4ch
0116  B0 4C                             mov     al,4ch
0118  CD 21                             int     21h
                                ;
011A                            cseg    ends
                                ;
                                        end     start
```

"crlfout","hexout","dump"を外部サブルーチンとして定義する

PSPのセグメント・アドレスを表示する

PSPの先頭から100Hバイトを表示する

リターンコード4CHを転送してプロセスを終了する

```
                                ;-----------------------------------------------;
                                ;   Function  4bh: Load and Execute a program  ;
                                ;             4dh: Retrieve the Return Code     ;
                                ;                                  of a Child   ;
                                ;-----------------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
```

```
                                ;
0100  BC 02A5 R                 start:  mov     sp,offset top_mem       スタック・ポインタを設定する

0103  B4 09                             mov     ah,9
0105  BA 0182 R                         mov     dx,offset prompt
0108  CD 21                             int     21h
010A  B4 0A                             mov     ah,0ah
010C  BA 01C5 R                         mov     dx,offset path
010F  CD 21                             int     21h
0111  8A 1E 01C6 R                      mov     bl,path[1]
0115  32 FF                             xor     bh,bh
0117  C6 87 01C7 R 00 90                mov     path[2][bx],0

011D  8C D8                             mov     ax,ds
011F  8E C0                             mov     es,ax
0121  BB 02A5 R                         mov     bx,offset top_mem
0124  B1 04                             mov     cl,4
0126  D3 EB                             shr     bx,cl
0128  43                                inc     bx
0129  B4 4A                             mov     ah,4ah
012B  CD 21                             int     21h
012D  72 33                             jc      error

012F  8C D8                             mov     ax,ds
0131  A3 01BB R                         mov     block[4],ax
0134  A3 01BF R                         mov     block[8],ax
0137  A3 01C3 R                         mov     block[12],ax

013A  B4 4B                             mov     ah,4bh
013C  BA 01C7 R                         mov     dx,offset path[2]
013F  BB 01B7 R                         mov     bx,offset block
0142  1E                                push    ds
0143  07                                pop     es
0144  B0 00                             mov     al,0
0146  CD 21                             int     21h
0148  72 18                             jc      error

014A  B4 4D                             mov     ah,4dh
014C  CD 21                             int     21h
014E  BB 01AA R                         mov     bx,offset retcode[2]
0151  E8 016B R                         call    convert
0154  8A C4                             mov     al,ah
0156  BB 01A8 R                         mov     bx,offset retcode
0159  E8 016B R                         call    convert

015C  BA 0197 R                         mov     dx,offset msgrtc
015F  EB 04 90                          jmp     msgout

0162  BA 01AD R                 error:  mov     dx,offset msgerr        エラーメッセージ

0165  B4 09                     msgout: mov     ah,9
0167  CD 21                             int     21h                     メッセージの表示

0169  CD 20                             int     20h                     プログラムの終了

016B  50                        convert:push    ax
016C  B1 04                             mov     cl,4
016E  D2 C8                             ror     al,cl
0170  E8 0174 R                         call    half
0173  58                                pop     ax
0174  24 0F                     half:   and     al,0fh
0176  04 30                             add     al,30h
0178  3C 3A                             cmp     al,3ah
017A  72 02                             jb      half1
017C  04 07                             add     al,7
017E  88 07                     half1:  mov     [bx],al
0180  43                                inc     bx
0181  C3                                ret
```

ロードして実行するプログラムのパス名を"Path"内に入力する，入力した文字列の最後に００Ｈを入れＡＳＣＩＺ文字列とする

このプログラムのメモリブロックの大きさを"top_mem"が含まれるまでの大きさのパラグラフ数でセットする，起動した時点ではプログラムにメモリブロックがメモリの最後まで割り当てられているのでロードに必要な空きブロックを確保する，メモリブロックが切り詰められなければ"error"へジャンプする

５ＣＨ，６ＣＨのデフォルトＦＣＢと８０Ｈのコマンドラインに与えるデータをこのプログラムと同じデータにセットする

"プログラムのロードと実行"のシステムコールロード，実行が正しく行われなければ"error"へジャンプする

"子プロセスの終了コードの読出し"のシステムコール

ＡＸレジスタに返されるリターンコードを表示する

```
                                        ;
0182  0D 0A 45 78 65 63                 prompt  db  0dh,0ah,'Execute file name $'
      75 74 65 20 66 69
      6C 65 20 6E 61 6D
      65 20 24
0197  0D 0A 52 65 74 75                 msgrtc  db  0dh,0ah,'Return code is '
      72 6E 20 63 6F 64
      65 20 69 73 20
01A8  30 30 30 30 24                    retcode db  '0000$'
01AD  0D 0A 45 72 72 6F                 msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
01B7  0000 0080 ???? 005C               block   dw      0,80h,?,5ch,?,6ch,?
      ???? 006C ????
01C5  3E 3F [                           path    db      62,63 dup (?)
            ??
               ]

                                        ;
0205     50 [                           buf     dw      80 dup (?)
            ????
                ]

                                        ;
02A5                                    top_mem label   word
                                        ;
02A5                                    cseg    ends
                                        ;
                                                end     start
```

Fig. 7.65　Sample List　31H, 4BH, 4CH, 4DH

実行手順　　FUNC−4B〈任意のパラメータ〉↵

を実行すると，

Execute file name

と，起動するプロセスを聞いてくるので，

FUNC−31 ↵
FUNC−4C ↵

と入力する．

```
A>func-4b filename.1 filename.2 ↵……"FUNC-4B.COM" を実行させる．
                                コマンドラインのパラメータはそのまま実行するプログラムに渡す
Execute file name func-31.com ↵……"FUNC-31.COM" をプログラム中からロードして実行させる
127E……PSPのセグメントアドレスを表示する．このアドレス以後にプログラムがロードされている
CD 20 00 40 00 9A F0 FF 0D F0 48 01 4B 12 BD 01   ^ .@......H.K.ｽ.   親プロセスと同じ環境
76 11 BA 04 76 11 4B 12 01 01 01 00 02 FF FF FF   v.ｺ.v.K.........   のセグメント・アドレ
FF FF FF FF FF FF FF FF FF FF FF FF 77 12 D4 FF   ............w.ﾔ.   スが渡されている
7E 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ~...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
CD 21 CB 00 00 00 00 00 00 00 00 00 46 49 4C      ^!ﾋ.........FIL   FCBにセットされた
45 4E 41 4D 45 31 20 20 00 00 00 00 00 46 49 4C   ENAME1  .....FIL   ファイル名
45 4E 41 4D 45 32 20 20 00 00 00 00 00 00 00 00   ENAME2  ........
17 20 66 69 6C 65 6E 61 6D 65 2E 31 20 66 69 6C   . filename.1 fil
65 6E 61 6D 65 2E 32 20 0D 65 78 65 63 0D 20 6D   ename.2 .exec. m
61 69 6C 2E 0D 0D 65 74 74 65 72 20 2D 2D 2D 0D   ail...etter ---.   "FUNC-4B.
7C 82 50 82 55 82 52 82 53 0D 00 00 00 00 00 00   !.P.U.R.S.......   COM" 起動時にコ
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................   マンドラインに与え
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................   たパラメータがセッ
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................   トされている
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
Return code is 0331 ……常駐終了したのでAH＝03Hとなりリターンコードは31Hとなっている
A>func-4b filename.1 filename.2 ……再び同じパラメータを与えて"FUNC-4B.COM"を実行

Execute file name func-4c.com ……こんどは"FUNC-4C.COM"を実行する
22AB ……"FUNC-31.COM"が常駐したので"FUNC-4C.COM"のロードされる領域がメモリの後方になっている
CD 20 00 40 00 9A F0 FF 0D F0 48 01 7F 22 BD 01    ^ .@.....H..'ｽ.
76 11 BA 04 76 11 7F 22 01 01 01 00 02 FF FF FF    v.ｺ.v..'........
FF FF FF FF FF FF FF FF FF FF 4B 12 D4 FF          ..........K.ﾖ.
AB 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ｫ".............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
CD 21 CB 00 00 00 00 00 00 00 00 00 46 49 4C       ^!ﾋ.........FIL
45 4E 41 4D 45 31 20 20 00 00 00 00 46 49 4C       ENAME1  ....FIL
45 4E 41 4D 45 32 20 20 00 00 00 00 00 00 00       ENAME2  .......
17 20 66 69 6C 65 6E 61 6D 65 2E 31 20 66 69 6C    . filename.1 fil
65 6E 61 6D 65 2C 32 20 0D 65 78 65 63 0D 20 6D    ename.2 .exec. m
61 69 6C 2E 0D 0D 65 74 74 65 72 20 2D 2D 2D 0D    ail...etter ---.
7C 82 50 82 55 82 52 82 53 0D 00 00 00 00 00 00    ¦.P.U.R.S.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................

Return code is 004C ……正常終了したのでAH＝00Hとなり，リターンコードは4CHとなっている
A>func-4b filename.1 filename.2

Execute file name func-4c.com ……再び"FUNC-4C.COM"を実行
22AB ……この前に実行した"FUNC-4C.COM"は常駐していないのでロードされるアドレスは同じ
CD 20 00 40 00 9A F0 FF 0D F0 48 01 7F 22 BD 01    ^ .@.....H..'ｽ.
76 11 BA 04 76 11 7F 22 01 01 01 00 02 FF FF FF    v.ｺ.v..'........
FF FF FF FF FF FF FF FF^C

Return code is 0100 ……CTRL-Cによる終了なのでAH＝01Hとなる．リターンコードは送られないので00Hである
A>■
```

Fig. 7.66　実行結果　31H，4BH，4CH，4DH

常駐したまま終了した場合，次にロードしたプログラムはそれ以後のメモリ領域が使用されるようになります．

FUNCTION　4EH, 4FH

ファンクション4EH　(Find Match File)――一致するファイルの検索

コール手順

AH　　←4EH
DS：DX　←パス名の先頭アドレス
CX　　←ファイルの属性
INT　21H

Return

キャリーがセットされた場合

AX＝02H　パス名が無効であり，該当するファイルが存在しない

AX＝12H　これ以上ファイルが存在しない

キャリーがセットされない場合

エラーなし

解説

DS：DXで渡されたパス名〔パス名の最終要素(ファイル名)にはワイルドカードを用いてもよい〕および属性に該当する最初のファイルを検索し，現在のDTA内に以下のフォーマットによる情報が書き込まれる．

DTA＋00H	検索するファイルの属性
DTA＋01H	ドライブ番号(0：A　1：B……)
DTA＋02H～＋09H	検索するパス名の中のファイル名
DTA＋0AH～＋0CH	検索するパス名の中の拡張子
DTA＋0DH～＋14H	システムで使用
DTA＋15H	検索されたファイルの属性
DTA＋16H～＋17H	time
DTA＋18H～＋19H	date
DTA＋1AH～＋1DH	ファイルの大きさ(4バイト)
DTA＋1EH～＋2AH	パックネーム(packed name)(ASCIZ文字列) (ファイル名が8文字未満の場合に，スペースが除かれたもの)

例：パックネーム　ＡＢＣ（名前）．ＸＹＺ（拡張子）→ＡＢＣ．ＸＹＺ 00H（packed name）

ファンクション4FH　(Step Through a Directory Matching Files)
——次に一致するファイルの検索

コール手順

```
AH　←4FH
INT　21H
```

Return

キャリーがセットされた場合

AX＝12H　これ以上ファイルが存在しない

キャリーがセットされない場合

エラーなし

解説

ディレクトリ内で次に一致するファイルを検索し，現在のDTA内にファンクション4EHと同じフォーマットによる情報が書き込まれる．現在のDTAのデータに基づいて検索が行われるので，DTAを変更した場合には，ファンクション4EHをコールしたときと同じアドレスをDTAにセットし直さなければならない．

FUNCTION 4EH, 4FH Sample Program

DIR コマンドの特定のファイル名の検索(DIR[〈ドライブ名：〉]〈パス名〉)に相当するプログラムを作成する．ファイル属性の指定は 17H とし，普通のファイルだけでなくディレクトリや隠されたファイルも表示させるようにする．

```
                                ;-----------------------------------------;
                                ;   Function  4eh: Find Match File        ;
                                ;             4fh: Step Through a Directory ;
                                ;                                Maching Files ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0080                                    org     80h
                                ;
0080    80 [                    cmdln   db      128 dup (?)
            ??
               ]

                                ;
0100                                    org     100h
                                ;
0100  B4 1A                     start:  mov     ah,1ah          ; デフォルトのDTAはパス名の入る"cmdln"と重なるので"dta"にDTAを変えておく
0102  BA 016A R                         mov     dx,offset dta
0105  CD 21                             int     21h

0107  8A 1E 0080 R                     mov     bl,cmdln[0]     ; パラメータがなにもなければ"nofile"へジャンプする
010B  84 DB                             test    bl,bl
010D  74 36                             je      nofile
010F  32 FF                             xor     bh,bh           ; パラメータの最後(CR)を00Hに変えASCIZ文字列とする
0111  C6 87 0081 R 00                   mov     cmdln[1][bx],0

0116  B4 4E                             mov     ah,4eh          ; "一致するファイルの検索"を行う
0118  BA 0082 R                         mov     dx,offset cmdln[2]
011B  B9 0017                           mov     cx,17h          ; 見つからなければ"nofile"へジャンプする
011E  CD 21                             int     21h
0120  72 23                             jc      nofile

0122  BB 0188 R                 nxtfile:mov     bx,offset dta[30]

0125  8A 17                     outlp:  mov     dl,[bx]         ; DTAの先頭アドレス+1EHからパック形式のファイル名がASCIZ文字列で格納されている
0127  84 D2                             test    dl,dl           ; このファイル名を00Hがくるまで出力する
0129  74 07                             je      next
012B  B4 02                             mov     ah,2
012D  CD 21                             int     21h
012F  43                                inc     bx
0130  EB F3                             jmp     outlp

0132  B4 09                     next:   mov     ah,9            ; CR, LF出力
0134  BA 0167 R                         mov     dx,offset msgcl
0137  CD 21                             int     21h             ; "次に一致するファイルの検索"を行う．新たに一致するファイルが見つかれば"next file"へ戻る
0139  B4 4F                             mov     ah,4fh
013B  CD 21                             int     21h
013D  73 E3                             jnc     nxtfile

013F  BA 015A R                         mov     dx,offset msged ; すべてのファイルを見つけたときのメッセージ
0142  EB 04 90                          jmp     msgout

0145  BA 014E R                 nofile: mov     dx,offset msgnof ; ファイルがないときのメッセージ

0148  B4 09                     msgout: mov     ah,9            ; メッセージの出力
014A  CD 21                             int     21h

014C  CD 20                             int     20h             ; プログラムの終了
```

```
                                        ;
014E  0D 0A 4E 6F 20 46                 msgnof  db  0dh,0ah,'No File',0dh,0ah,'$'
      69 6C 65 0D 0A 24
015A  0D 0A 45 6E 64 20                 msged   db  0dh,0ah,'End File',0dh,0ah,'$'
      46 69 6C 65 0D 0A
      24
0167  0D 0A 24                          msgcl   db  0dh,0ah,'$'
                                        ;
016A     80 [                           dta     db      128 dup (?)
           ??
               ]

                                        ;
01EA                                    cseg    ends
                                        ;
                                                end     start
```

Fig. 7.67　Sample List　4EH, 4FH

実行手順　　FUNC−4E　[〈ドライブ名：〉]〈パス名〉 ↵

```
A>dir ↵……ルート・ディレクトリの内容を確認する

 ドライブ A: のボリュームラベルは MS-DOS HB
 ディレクトリは A:¥

COMMAND  COM    17190  83-09-19  16:19
BIN          <DIR>      84-02-22  20:32
COM          <DIR>      84-02-22  20:21
AUTOEXEC BAT       61   84-02-25  17:22
CONFIG   SYS       70   84-03-20  21:07
FUNC         <DIR>      84-02-22  22:04
FACT     EXE    13496  84-03-05  19:14
TEST             160   84-02-29  21:21
SUBDIR1      <DIR>      84-02-23   1:19
CAL          <DIR>      84-02-23   1:40
TXT          <DIR>      84-02-23   5:07
ASMWORK      <DIR>      84-02-25   7:22
      12 個のファイルがあります
  715776 バイトが使用可能です

A>func-4e *.* ↵……検索する属性に17Hを指定したのでボリュームラベルを除く，すべてのディレクトリ・エントリを検索する
IO.SYS            隠されたファイル("IO.SYS","MSDOS.SYS")やディレクトリ(BIN, COMなど)も検索されている
MSDOS.SYS
COMMAND.COM
BIN
COM
AUTOEXEC.BAT
CONFIG.SYS
FUNC              ファイル名の表示はバックネームの形式である
FACT.EXE
TEST
SUBDIR1
CAL
TXT
ASMWORK

End File

A>func-4e bin¥..¥*.sys ↵……拡張子がSYSのファイルをルート・ディレクトリから検索する
IO.SYS                    検索にはワイルドカードおよびパス名が使用できる
MSDOS.SYS         3つのファイルがあった
CONFIG.SYS

End File

A>■
```

Fig. 7.68　実行結果　4EH, 4FH

ドライブ名を指定しなければカレント・ドライブ，ディレクトリ名を指定しなければカレント・ディレクトリを検索することがわかります．

ファイル名の途中にスペースがあると正しく表示されません(スペース以後は，パックネームにする際に無視される)．これは，パス名にはスペースが使えないことを考えれば当然でしょう．

FUNCTION 56H

ファンクション56H (Move a Directory Entry)——ディレクトリ・エントリの移動

コール手順

```
AH      ← 56H
DS：DX  ←既存のファイルのパス名の先頭アドレス
ES：DI  ←新規のファイルのパス名の先頭アドレス
INT 21H
```

Return

キャリーがセットされた場合

AX＝02H　DS：DX で指定されたファイルが存在しない

AX＝03H　パス名が無効であるか，またはパスが存在しない

AX＝05H　DS：DX で指定されたパス名がディレクトリであったか ES：DI で指定されたファイルがすでに存在している

AX＝11H　既存のパス名と新規のパス名は異なったドライブに存在する

キャリーがセットされない場合

エラーなし

解説

異なるディレクトリ間でのファイルの RENAME に相当する．もちろん，同一ディレクトリ内での RENAME の役割も果たす．このシステムコールにより，あるディレクトリにファイルが移動されたかのように見える．このとき，ワイルドカードは使用できない．

FUNCTION 56H　Sample Program

プログラム中で，既存のファイルと新規のファイルのパス名を与え，パス名を変えるプログラムを作成する．

```
                                ;-----------------------------------------;
                                ;   Function  56h: Move a Directory Entry ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0100                                    org     100h
                                ;
0100  B4 09                     start:  mov     ah,9
0102  BA 015F R                         mov     dx,offset msgodpt
0105  CD 21                             int     21h
0107  B4 0A                             mov     ah,0ah
0109  BA 0185 R                         mov     dx,offset oldpath
010C  CD 21                             int     21h
010E  8A 1E 0186 R                      mov     bl,oldpath[1]
0112  32 FF                            xor     bh,bh
0114  C6 87 0187 R 00 90                mov     oldpath[2][bx],0

011A  B4 09                             mov     ah,9
011C  BA 0172 R                         mov     dx,offset msgnwpt
011F  CD 21                             int     21h
0121  B4 0A                             mov     ah,0ah
0123  BA 01C5 R                         mov     dx,offset newpath
0126  CD 21                             int     21h
0128  8A 1E 01C6 R                      mov     bl,newpath[1]
012C  32 FF                             xor     bh,bh
012E  C6 87 01C7 R 00 90                mov     newpath[2][bx],0

0134  B4 56                             mov     ah,56h
0136  BA 0187 R                         mov     dx,offset oldpath[2]
0139  BF 01C7 R                         mov     di,offset newpath[2]
013C  1E                                push    ds
013D  07                                pop     es
013E  CD 21                             int     21h
0140  BA 0155 R                         mov     dx,offset msgerr
0143  72 03                             jc      error
0145  BA 014E R                         mov     dx,offset msgok

0148  B4 09                     error:  mov     ah,9
014A  CD 21                             int     21h

014C  CD 20                             int     20h
                                ;
014E  0D 0A 4F 6B 0D 0A         msgok   db  0dh,0ah,'Ok',0dh,0ah,'$'
      24
0155  0D 0A 45 72 72 6F         msgerr  db  0dh,0ah,'Error',0dh,0ah,'$'
      72 0D 0A 24
015F  0D 0A 4F 6C 64 20         msgodpt db  0dh,0ah,'Old path name ? $'
      70 61 74 68 20 6E
      61 6D 65 20 3F 20
      24
0172  0D 0A 4E 65 77 20         msgnwpt db  0dh,0ah,'New path name ? $'
      70 61 74 68 20 6E
      61 6D 65 20 3F 20
      24
0185  3E 3F [                   oldpath db  62,63 dup (?)
            ??
               ]

01C5  3E 3F [                   newpath db  62,63 dup (?)
            ??
               ]

                                ;
0205                            cseg    ends
                                ;
                                        end     start
```

パス名の変更を行おうとするファイル名を〝oldpath〟内に入力する
入力は61バイトまで可能である
入力された文字列の最後を00Hとし，ASCIZ文字列とする

新規のパス名を〝new path〟内に入力する

DS：DXで変更を行うパス名の先頭アドレス，ES：DIで新規のパス名の先頭アドレスを指定しパス名の変更を行う

変更できれば〝Ok〟と表示する
すでに新規のパス名と同一のファイルが存在していたり，パス名が無効な場合は〝Error〟と表示する

プログラムの終了

Fig. 7.69　Sample List　56H

実行手順　　FUNC－56↵

```
A>dir subdir1↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1

.            <DIR>      84-03-24  12:32
..           <DIR>      84-03-24  12:32
SUBDIR2      <DIR>      84-03-24  12:32
FUNC-56  ASM       965  84-03-01  15:25
       4 個のファイルがあります
   260096 バイトが使用可能です

A>dir subdir1¥subdir2↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1¥subdir2

.            <DIR>      84-03-24  12:32
..           <DIR>      84-03-24  12:32
       2 個のファイルがあります
   260096 バイトが使用可能です
```

"FUNC-56．ASM" がディレクトリ "¥SUBDIR1" にありディレクトリ "¥SUBDIR1¥SUBDIR2" にはファイルがないことを確認しておく

```
A>func-56↵ ……"FUNC-56．COM" の実行

Old path name ? subdir1¥func-56.asm↵ ……古いパス名を入力
New path name ? subdir1¥subdir2¥56h.asm↵ ……新たなパス名を入力
Ok ……正常終了したことを示すメッセージ

A>dir subdir1↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1

.            <DIR>      84-03-24  12:32
..           <DIR>      84-03-24  12:32
SUBDIR2      <DIR>      84-03-24  12:32
       3 個のファイルがあります
   260096 バイトが使用可能です

A>dir subdir1¥subdir2↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥subdir1¥subdir2

.            <DIR>      84-03-24  12:32
..           <DIR>      84-03-24  12:32
56H      ASM       965  84-03-01  15:25
       3 個のファイルがあります
   260096 バイトが使用可能です

A>■
```

ディレクトリ "¥SUBDIR1" にあった "FUNC-56．ASM" がディレクトリ "¥SUBDIR1¥SUBDIR2" の "56H．ASM" というファイルになっていることがわかる

Fig. 7.70　実行結果　56H

このシステムコールは，新たにディレクトリを作成するわけではないので，それにパス名が制約を受けます．

FUNCTION　57H

ファンクション57H　(Get／Set Date／Time of File)
——ファイルの時間情報の読出し／設定

コール手順

日付／時刻を得る場合

```
AH　←57H
AL　←00H
BX　←ファイル・ハンドル
INT　21H
```

日付／時刻をセットする場合

```
AH　←57H
AL　←01H
BX　←ファイル・ハンドル
CX　←セットする時刻
DX　←セットする日付
INT　21H
```

Return

キャリーがセットされた場合

AX＝01H　無効なファンクション(ALが00H～01Hの範囲外の場合)

AX＝06H　無効なファイル・ハンドル．BXに渡されたファイル・ハンドルは現在オープンされていない

キャリーがセットされない場合

AL＝00Hの場合　DX／CXにこのファイル・ハンドルが最後に編集された日付／時刻を返す

AL＝01Hの場合　エラーなく日付／時刻がセットされた．この日時はファイルがクローズされるまで記録されない

解説

ファイルが最後に編集された日付／時刻を得たり，セットすることができる．日付／時刻をセットする場合，ファイルがクローズされるまで日付／時刻は記録されない．DX，CXレジスタの日付／時刻は，FCBにセットされるものと同じフォーマットとなる．

FUNCTION　57H　Sample Program

ファイルをオープン(ファンクション3DH)して，ファイルの編集年を1年進めるプログラムを作成する．

```
                                ;-----------------------------------------;
                                ;   Function  57h: Get/Set Date/Time of File  ;
                                ;-----------------------------------------;
                                ;
0000                            cseg    segment
                                        assume  cs:cseg,ds:cseg
                                ;
0080                                    org     80h
                                ;
0080    80 [                    cmdln   db      128 dup (?)
            ??
               ]

                                ;
0100                                    org     100h
                                ;
0100  8A 1E 0080 R              start:  mov     bl,cmdln[0]     ; パラメータが何もないなら"error"へジャンプする
0104  84 DB                             test    bl,bl
0106  74 3D                             je      error
0108  32 FF                             xor     bh,bh           ; パラメータの最後（CR）を00Hに変え，ASCIZ文字列とする
010A  C6 87 0081 R 00                   mov     cmdln[1][bx],0

010F  B4 3D                             mov     ah,3dh          ; コマンドラインのパラメータで指定されたファイルを読出しモードでオープンする
0111  BA 0082 R                         mov     dx,offset cmdln[2]  オープンできなければ"error"へジャンプする
0114  B0 00                             mov     al,0
0116  CD 21                             int     21h
0118  72 2B                             jc      error
011A  A3 0157 R                         mov     handle,ax

011D  B4 57                             mov     ah,57h          ; オープンしたファイルの日付と時刻を得る
011F  B0 00                             mov     al,0            エラーならば"error"へジャンプする
0121  8B 1E 0157 R                      mov     bx,handle
0125  CD 21                             int     21h
0127  72 1C                             jc      error
0129  D1 EA                             shr     dx,1            ; 日付の年度を1年進める
012B  FE C6                             inc     dh
012D  D1 D2                             rcl     dx,1
012F  B4 57                             mov     ah,57h          ; ファイルの日付と時刻をセットする
0131  B0 01                             mov     al,1            エラーならば"error"へジャンプする
0133  8B 1E 0157 R                      mov     bx,handle
0137  CD 21                             int     21h
0139  72 0A                             jc      error

013B  B4 3E                             mov     ah,3eh          ; オープンしたファイルをクローズしておく
013D  8B 1E 0157 R                      mov     bx,handle       クローズできれば"return"へジャンプする
0141  CD 21                             int     21h
0143  73 07                             jnc     return

0145  B4 09                     error:  mov     ah,9            ; 何らかのエラーがあったと表示する
0147  BA 014E R                         mov     dx,offset msgerr
014A  CD 21                             int     21h

014C  CD 20                     return: int     20h             ; プログラムの終了
                                ;
014E  0D 0A 45 72 72 6F         msgerr  db      0dh,0ah,'Error $'
      72 20 24
0157  ????                      handle  dw      ?
                                ;
0159                            cseg    ends
                                ;
                                        end     start
```

Fig. 7 .71 Sample List 5 7 H

実行手順　　DIR コマンドでファイルの日付を確認しておいてから
FUNC−57 〈パス名〉 ↵
を実行して再び DIR コマンドでファイルの日付が 1 年進んでいることを確認する

```
A>dir func-57.asm↵ ……"FUNC-57.ASM" の日付を確認しておく

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

                                  ┌──84年2月28日となっている
FUNC-57  ASM       896  84-02-28  20:42
       1 個のファイルがあります
  329728 バイトが使用可能です

A>func-57 func-57.asm↵ ……"FUNC-57.COM" を実行する

A>dir func-57.asm↵ ……"FUNC-57.ASM" の日付が変更されたか確かめる

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

                                  ┌──85年2月28日となった
FUNC-57  ASM       896  85-02-28  20:42
       1 個のファイルがあります
  329728 バイトが使用可能です

A>■
```

Fig. 7.72　実行結果　57H

ファイルの日付の管理は MS-DOS が自動的に処理するので，日付を変更することは実際には必要ありませんが，ファイルをコピーするときに，その日付も保存したい場合には有効です．

8章　高級言語の紹介

ソフトウェアの開発目的には，商業ベース，実験，研究，大学でのプログラム演習と様々なものが考えられますが，こういったソフトウェアの開発手段としてもパーソナル・コンピュータは有効に利用することができます．もちろん大型コンピュータでの開発環境と比較すれば，処理速度，処理系の仕様，ユーティリティの充実といった面で劣っていますが，ひと昔前のミニコンクラスと比較すれば，最近の16ビットマシンの方が総合的な能力では優れているといえるでしょう．また，パーソナル・コンピュータの低コスト，インタラクティブな操作といったメリットを考えれば，その開発環境のポータビリティは十分利用する価値があります．

MS-DOSでは現在次に示すような高級言語を利用することができますが，その種類，仕様ともパーソナル・コンピュータのソフトウェア環境としてはかなり充実したものといえます．

BASIC, FORTRAN, COBOL, PASCAL, PL／I
C, ADA, PROLOG, LISP, APL, LOGO, FORTH

本章では，上に示した言語のうち，パーソナル・コンピュータのユーザーにとって最もなじみ深いBASIC(MS-BASCOM)と，システム記述言語として高い評価を受け，近年急速にユーザーが増加しているC(Lattice C)を取り上げ，実際のプログラムを通してその言語仕様および開発手順を示しています．なお，本書では各言語に関する詳細な説明はしていませんが，パーソナル・コンピュータの環境で利用できる言語について興味のある方は，『パソコン言語学』（アスキー発行　石田晴久 監修 他共著）において言語の特徴，沿革，仕様等について詳しく解説されているので，詳細については同書を参照されることをお勧めします．

8.1 C言語

CはUNIXを記述している言語として有名であるように，システム記述言語と呼ばれており，アセンブラに近い記述をすることも可能な言語です．Cで記述された身近な例としては，MS-DOSのマクロアセンブラ，作表言語のMultiplanなどが挙げられますが，言語として以下の特徴を備えているために，アセンブリ言語の代わりに使うユーザーが増えてきています．

(1)豊富な演算子によりアセンブラに近い記述が可能
(2)柔軟なポインタ操作が可能

(3)再帰呼出しが可能

(4)テキスト置換を主体としたマクロ機能

現在MS-DOSで実行可能なCコンパイラが何種類か販売されていますが，Cには標準となる規格が定まっているわけではなく，UNIX上で使用されるCの言語仕様が事実上の標準となっています．本章で紹介するCコンパイラは，Lattice社からリリースされているものですが，この処理系の仕様もUNIX ver. 7コンパチブルとなっています．

8.1.1　Lattice Cの特徴

Lattice Cは8086CPUの機械語を生成するCコンパイラであり，現在ver. 2.00がリリースされています．旧バージョン(ver. 1.04まで)ではプログラムとデータ領域がそれぞれ64Kバイト以内のメモリモデル(Sモデル)しかサポートされておらず，8086CPUの1Mバイトのメモリ空間を有効に利用することができませんでしたが，ver. 2.00ではプログラム領域とデータ領域の大きさにより，4つのメモリモデル(S, P, D, Lモデル)を扱うことができます．そのため旧バージョンの"C.OBJ"，"LC．LIB"は，メモリモデルごとに分けられ，Table．8．1に示す4組が供給されています．

Table　8．1　Lattice CのメモリモデルとOBJ，LIBファイル

データ領域＼プログラム領域	64Kバイト以内	1Mバイト以内
64Kバイト以内	Sモデル CS．OBJ LCS．LIB	Pモデル CP．OBJ LCP．LIB
1Mバイト以内	Dモデル CD．OBJ LCD．LIB	Lモデル CL．OBJ LCL．LIB

これらのメモリモデルは"LC1．EXE"でパス1を行うときに，オプションとして次のように指定するようになっています．

書式　LC1 〈ソースファイル名〉[〈オプション〉]

〈オプション〉＝-ms, -mp, -md, -ml

また，これらのメモリモデルによらず同一形式でアセンブラのリンクモジュールを作成できるように，メモリモデルに応じたマクロ定義のファイルが用意されています．さらに，ver. 2．00ではMS-DOS ver. 2.0に対応する機能(パス名のサポート，ERRORLEVELに終了時に値を返すなど)が追加されているほか，できあがったオブジェクトにはスタックオーバーフローのチェックを行う機能も付け加わっています．

8.1.2 Lattice C サンプル・プログラム

Lattice C の文法と使用例を，実際のプログラムを通して見ていくことにします．ここで紹介するサンプル・プログラムは，コマンドラインのパラメータで与えられたファイルを，ディスク上のすべてのディレクトリから捜し出し，ファイルが見つかったディレクトリを表示するものです．ディレクトリの検索には，MS-DOS のシステムコールを利用していますが，Lattice C では MS-DOS のシステムコールを直接呼び出す記述ができるため，機械語のプログラムを作成する必要がありません．

List8．1にサンプル・プログラム〝WHEREIS．C〟を示します．

List 8．1 WHEREIS．C

```
/*
    Whereis : Where is the desired file ?
*/

#include <stdio.h>
#include <ctype.h>
#include <dos.h>

#define FALSE 0
#define TRUE  1

char    path[192];
char    fname[128];
int     exist;
union   REGS inregs, outregs;

main (argc, argv)
int     argc;
char    *argv[];
{
        int     i, level, nmptr;
        char    *ptr;

        if (argc == 1)
                printf ("Usage : whereis [<file list>]¥n");
        else
        {
                for (i = 1; i < argc; i++)
                {
                        strcpy (fname, argv[i]);
                        ptr = fname;
                        while (*ptr = toupper (*ptr)) ptr++;
                        printf ("%s¥n", fname);

                        level = nmptr = 0;
                        if (argv[i][1] == ':')
                        {
                                strcpy (&path[nmptr], fname);
                                nmptr += 2;
                                strcpy (fname, &path[nmptr]);
                        }
                        path[nmptr++] = '¥¥';

                        exist = FALSE;
                        find (level+1, nmptr);
                        if (!exist) printf ("  Not found¥n");
                        printf ("¥n");
                }
        }
}
```

コメントは〝/*〟と〝*/〟で囲む

指定されたファイルをこの場所に挿入してコンパイルする

FALSEを0，TRUEを1に置き換えてコンパイルする

パス名を格納する文字型変数の配列

変数の定義

Cではメインプログラムにmainという名前を付ける

プログラムに与えられたアーギュメントの個数，1ならアーギュメントなし

与えられたアーギュメントのポインタの配列

main内で使用する変数の定義

アーギュメントがない場合には使用法を表示する

アーギュメントがある場合

アーギュメントの数だけ繰り返す

fnameにi番目のアーギュメントargv[i]をコピーする

fname内の小文字を大文字に変換して表示する

初期設定

ドライブ名が指定されていればfnameのドライブ名をpathに移す

最初はルート・ディレクトリから捜しはじめる

fnameの名前のファイルをすべてのディレクトリから捜す

ファイルが見つからない場合のメッセージ

改行する

```
find (level, nmptr)
int     level, nmptr;
{
        int     endptr;
        char    dta[44];……ディレクトリ情報が返されるＤＴＡ領域を確保する

        chkexist (nmptr);……まず，このディレクトリに目的のファイルがあるかチェックする

        strcpy (&path[nmptr], '*.*');……次のこのディレクトリ内のすべてのファイルを捜す
        if (!searchfirst (dta))……はじめのファイルが見つかれば以下を実行する
        {
                do
                {
                        if ((dta[21] & 0x10) &&
                            strcmp (&dta[30], '.') &&
                            strcmp (&dta[30], '..'))
                        {
                                strcpy (&path[nmptr], &dta[30]);
                                endptr = nmptr + strlen (&dta[30]);
                                path[endptr++] = '¥¥';
                                find (level+1, endptr);……再帰呼出し
                        }
                } while (!searchnext (dta));……次のファイルが見つからなくなるまで繰り返す
        }
}

chkexist (nmptr)
int     nmptr;
{
        char    dta[44];……ＤＴＡ領域を確保する

        strcpy (&path[nmptr], fname);
        if (!searchfirst (dta))
        {
                path[nmptr] = '¥0';
                printf("  Find in %s¥n",path);
                exist = TRUE;
        }
}

searchfirst (dta)
char    *dta;
{
        setdta (dta);……ＤＴＡのセット
        inregs.x.ax = 0x4e00;
        inregs.x.cx = 0x0010;
        inregs.x.dx = &path[0];
        intdos (&inregs, &outregs);
        return (outregs.h.al);
}

searchnext (dta)
char    *dta;
{
        setdta (dta);……ＤＴＡのセット
        inregs.x.ax = 0x4f00;
        intdos (&inregs, &outregs);
        return (outregs.h.al);
}

setdta (dta)
char    *dta;
{
        inregs.h.ah = 0x1a;
        inregs.x.dx = dta;
        intdos (&inregs, &outregs);
}
```

ディレクトリ・エントリがディレクトリ属性を持ち"."または".."ディレクトリでないならそのディレクトリに移って検索を続ける

見つけたディレクトリ名をpathにセット

pathとfnameをつなげたASCIZ文字列のパス名が存在すれば，そのディレクトリ名を
Find in〈ディレクトリ名〉
と表示する

最初のディレクトリ・エントリの検索のシステムコール
属性は１０Ｈを指定しディレクトリと通常のファイルのみを検索する
またドライブはカレント・ドライブとする

次のディレクトリ・エントリの検索のシステムコール
ディレクトリ・エントリが見つかればＡＬレジスタの値００Ｈは変化せず見つからなければ１２Ｈとなるのでキャリーフラグを調べるかわりに使用する

ＤＴＡのセットのシステムコール

次に〝WHEREIS．C〟をコンパイルして，実行形式のファイル〝WHEREIS．EXE〟を作成します．Lattice Cは2パス形式のコンパイラであり，〝LC1．EXE〟で中間コードファイルを作成し，〝LC2．EXE〟で8086CPUのオブジェクトファイル〝WHEREIS．OBJ〟を作成します．次にリンカでCのライブラリとリンクするときに，メモリモデルにSモデルを用いているので〝CS．OBJ〟と〝LCS．LIB〟をリンクして実行形式のファイルを作成しています（Fig. 8.1).

```
A>dir whereis↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

WHEREIS  C        1916  84-03-18  16:29 ……Cのソースファイル(拡張子は〝．C〟)
        1 個のファイルがあります
   367616 バイトが使用可能です

A>lc1 whereis -ib  ……コンパイラのパス1を起動する．〝-ib〟はファイルをドライブBからincludeせよという指示
Lattice C Compiler (Phase 1) V2.00
Copyright (C) 1982 Lattice, Inc.

A>dir whereis↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

WHEREIS  C        1916  84-03-18  16:29
WHEREIS  Q        4811  84-03-18  16:30 ……パス1で作られた中間ファイル
        2 個のファイルがあります
   362496 バイトが使用可能です

A>lc2 whereis↵ ……コンパイラのパス2を起動する
Lattice 8086 Code Generator V2.00
Copyright (C) 1982 Lattice, Inc.
Module size P=0320 D=0173 ……Pはプログラム，Dはデータ領域のサイズ(16進)

A>dir whereis↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

WHEREIS  C        1916  84-03-18  16:29
WHEREIS  OBJ      1910  84-03-18  16:30 ……オブジェクトファイル
        2 個のファイルがあります
   365568 バイトが使用可能です

A>link↵ ……リンカを起動する

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: b:cs whereis↵ ……スモールメモリモデルではまず〝CS．OBJ〟を先頭にリンクする必要がある
Run File [A:CS.EXE]: whereis↵ ……出力ファイルを指定する．指定しないと〝CS．EXE〟になってしまう
List File [NUL.MAP]: whereis↵ ……マップファイルの指定
Libraries [.LIB]: b:lcs↵ ……スモールメモリモデルのライブラリファイルとリンクする
```

また，PATHは〝A：¥；B：¥〟に設定しておいてコンパイラや〝．H〟ファイルはすべてドライブBに入れてあるメモリモデルを指定していないので，できあがるファイルはスモールメモリモデルとなる

```
A>dir whereis↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

WHEREIS  C        1916   84-03-18   16:29
WHEREIS  MAP       352   84-03-18   16:31 ……マップファイル
WHEREIS  OBJ      1910   84-03-18   16:30
WHEREIS  EXE     12488   84-03-18   16:31 ……実行型ファイル
        4 個のファイルがあります
   351232 バイトが使用可能です

A>■
```

Fig. 8 . 1　コンパイル手順

このプログラムを実行してみます.

```
A>whereis fact.c more.com↵ ……"WHEREIS . EXE" の実行. コマンドラインには空白で区切って
FACT.C                        いくつもファイル名が入力できる
  Find in ¥SUBDIR1¥ }
  Find in ¥CAL¥     } "FACT . C" が "¥SUBDIR1" と "¥CAL" で見つかったと表示される

MORE.COM
  Find in ¥COM¥


A>whereis schdir↵
SCHDIR
  Not found ……"SCHDIR" というファイルはどこにもなかった


A>whereis subdir*↵ ……パラメータにはワイルドカードも使用できる
SUBDIR*
  Find in ¥
  Find in ¥BIN¥
  Find in ¥SUBDIR1¥


A>■
```

Fig. 8 . 2　WHEREIS . EXE実行結果

Lattice Cは，8087演算プロセッサをサポートする予定のフルセット仕様のCの処理系であり，生成するオブジェクトの実行速度は現存する8086CPU用のCコンパイラの中では高速の部類にはいります．また，MS-DOS上ではライブラリの管理も十分に行えるため，本格的なプログラム開発にも十分耐え得るといえるでしょう．

8.2 BASIC

BASICが1964年に米Dartmouth大学で開発されたときは，学生にコンピュータを教えるための教育用言語として，大型コンピュータのTSSで使用されていました．それが現在では，パーソナル・コンピュータのメイン言語としての位置を占めるようになっています．

BASICのいちばんの特徴は，融通のきく構文体系にあります．その扱いやすさゆえ初心者向けの言語として多くのユーザーに受け入れられていますが，これは裏を返せば見通しの悪いプログラムになりやすいという欠点でもあります．しかし，BASICの開発環境には見逃せないメリットがあります．本章で取り上げたMS-BASICおよびMS-BASCOMの関係がそうであるように，インタープリタとコンパイラの2つの処理系を備えることによって，対話形式でプログラム開発を行い，コンパイラで実行速度を向上させるという手法がとれる点です．

8.2.1 MS-BASIC

MS-BASICは，MS-DOS上で走るマイクロソフト社製の8086CPU用のBASICインタープリタです．MS-BASICは，同社のBASIC-80をベースとして，コマンド，ステートメント，関数は基本的に同じ仕様になっていますが，CPUに依存する部分(DEF SEG, PEEK, POKEなど)は8086CPUのアーキテクチャに合せて変更されています．

現在パーソナル・コンピュータに搭載されているマイクロソフト系BASICは，BASIC-80の基本仕様に各機種ごとのハードウェア(グラフィックなど)をサポートする命令が拡張されたものですから，スタンドアローンBASICに慣れているユーザーであれば，支障なくMS-BASICを利用することができます．

8.2.2 MS-BASCOMの特徴

MS-BASCOMは，インタープリタとコンパイラの性格による仕様の相違を除けば，MS-BASICのすべての命令をサポートしたBASICコンパイラです．

インタープリタ形式のMS-BASICでプログラムの開発およびデバッグを行い，プログラムの動作を確認した後で，MS-BASCOMでコンパイルしてスピードアップを計ることができます．

8.2.3 MS-BASCOMサンプル・プログラム

MS-BASCOMのサンプル・プログラムとして，ディレクトリの木構造を表示するプログラムを示します．このプログラムの処理のうち，BASICではできないディレクトリの検索部分をアセンブラで作成し，残りの処理をBASICで行い，実行ファイルはコンパイルしたモジュールとアセンブラで生成したモジュールをリンクしています．

BASICから機械語のルーチンを呼ぶには，USR関数とCALL文がありますが，機械語のモジュールとリンクするには名前で呼び出せるCALL文を使用しなければなりません．

List 8.2とList 8.3がメイン・プログラムとなるソースファイル〝TINYTREE．BAS〟と機械語サブルーチン〝SCHDIR．ASM〟のリスティングファイルです．BASICのソースファイルは，MS-BASICなどのインタープリタのエディタで作成するとプログラムの編集作業が容易に行えますが，そういった環境がなければ，MS-DOS上で走るエディタで作成することになります．

List 8.2　TINYTREE.BAS

```
1000 '
1010 '    tiny tree command
1020 '
1030 ' $title: 'MS-DOS Handbook' $subtitle: 'Tiny tree' ……MS-BASCOMの疑似命令
1040 OPTION BASE 1
1050 DEFINT A-Z
1060 DIM START(4),LVLDIR$(3)
1070 DEF FNNAME$ = LEFT$(FILENAME$+" "+STRING$(12,"-"),13)
1080 DEF FNSTR$ = LEFT$(DUMMY$,INSTR(1,DUMMY$,CHR$(0))-1)
1090 NULFILE$ = STRING$(13," ")
1100 DUMMY$ = STRING$(64," ") + "" ……機械語ルーチンから文字列を受け取るための領域を準備する
1110 WILDCARD$ = "¥*.*"+CHR$(0)
1120 LEVEL = 0
1130 '
1140 PRINT : INPUT "Root";ROOT$ : PRINT                                    ディレクトリツリーのルートの部分に相当するディレクトリを入力する
1150 IF ROOT$ = "" THEN CALL CURRENTDIR(DUMMY$) : ROOT$ = "¥" + FNSTR$
1160 IF RIGHT$(ROOT$,1) = "¥" THEN ROOT$ = LEFT$(ROOT$,LEN(ROOT$)-1)
1170 IF INSTR(1,ROOT$,":") = LEN(ROOT$) THEN FILENAME$ = "¥" : GOTO 1250
1180 '
1190 PATH$ = ROOT$ + CHR$(0) ……ASCIZ文字列にする
1200 CALL SEARCHFIRST(PATH$,STATUS,LEVEL,DUMMY$)                           PATH$のディレクトリを検索見つからなければ1560行へジャンプ
1210 IF STATUS = -1 OR LEFT$(DUMMY$,1) = CHR$(0) THEN 1560
1220 FILENAME$ = FNSTR$
1230 IF (STATUS AND 2) = 0 THEN PRINT FILENAME$ : END……PATH$のディレクトリ・エントリがディレクトリでなければFILENAME$のみ表示して終了する
1240 '
1250 LINE$ = FNNAME$
1260 LEVEL = LEVEL + 1
1270 START(LEVEL) = 0
1280 FIRST = 0
1290 '
1300 WHILE LEVEL > 0 ……LEVELが正の間繰り返せ
1310   GOSUB 1600 ……表示する行を途中まで作成
1320   GOSUB 1690 ……検索するディレクトリのパス名をPATH$にセット
1330   IF START(LEVEL) <> 0 THEN 1370 ……このディレクトリはすでに検索したことがあるなら1370行へジャンプ
1340     CALL SEARCHFIRST(PATH$,STATUS,LEVEL,DUMMY$) ……最初のディレクトリ・エントリの検索
1350     BRANCH$ = "---"
1360     GOTO 1400
1370   IF START(LEVEL) <> 1 THEN LEVEL = LEVEL - 1 : GOTO 1530 ……このディレクトリの検索が終わったなら親ディレクトリの検索に移る
1380     CALL SEARCHNEXT(STATUS,LEVEL,DUMMY$) ……次のディレクトリ・エントリーの検索
1390     BRANCH$ = "|-"
1400     IF STATUS = -1 THEN 1560
1410     FILENAME$ = FNSTR$
1420     IF LEVEL = 4 AND ((STATUS AND 2) = 2)                             ディレクトリが4段以上作成されている場合はそれ以上検索せず頭に"¥"を付けてファイルと区別する
           THEN FILENAME$ = "¥" + FILENAME$ :
             STATUS = (STATUS AND 1) : BRANCH$ = LEFT$(BRANCH$,3)
1430     IF (STATUS AND 2) = 0                                             ディレクトリならば続きがあることを示す"----"を，表示するデータに加える
           THEN LINE$ = LINE$ + BRANCH$ + FILENAME$
           ELSE LINE$ = LINE$ + BRANCH$ + FNNAME$
1440     IF STATUS = 0 THEN START(LEVEL) = 1 : GOTO 1510 ……次のファイルがあるなら1510行へジャンプ
1450     IF STATUS = 1 THEN LEVEL = LEVEL -1 : GOTO 1510 ……ディレクトリにファイルがなくなったら親ディレクトリの検索に移る
1460     LVLDIR$(LEVEL) = FILENAME$
1470     IF STATUS = 3 THEN START(LEVEL) = 2                               検索したディレクトリ・エントリがディレクトリであったなら次のディレクトリの検索へ移る
           ELSE START(LEVEL) = 1
1480     LEVEL = LEVEL + 1
1490     START(LEVEL) = 0
1500   GOTO 1320 ……BASICでは再帰手続きが作れないため配列とGOTOで実現する
1510   PRINT LINE$                                                         ディレクトリツリーの1行表示ディレクトリにファイルがなくなった場合は1行"|"だけの行を表示
1520   IF (STATUS AND 1) = 1 THEN LEVEL = LEVEL + 1 : GOSUB 1600 :
                                  LEVEL = LEVEL - 1 : PRINT LINE$
1530 WEND
1540 END
1550 '
1560 PRINT "Sorry not found" : END ……入力したディレクトリ・エントリが見つからないときのメッセージ
1570 '
1580 '    make tree one line
1590 '
```

```
1600 IF FIRST = 0 THEN FIRST = 1
                  ELSE LINE$ = NULFILE$
1610 FOR I = 1 TO LEVEL - 1
1620     IF START(I) = 1 THEN LINE$ = LINE$ + "  |   "
                         ELSE LINE$ = LINE$ + "      "
1630     LINE$ = LINE$ + NULFILE$
1640 NEXT
1650 RETURN
1660 '
1670 '    make search pathname
1680 '
1690 PATH$ = ROOT$
1700 FOR I = 1 TO LEVEL - 1
1710   PATH$ = PATH$ + "¥" + LVLDIR$(I)
1720 NEXT
1730 PATH$ = PATH$ + WILDCARD$
1740 RETURN
```

ディレクトリツリーの表示用の1行分のデータを作成する

"〈ディレクトリ名〉¥〈ディレクトリ名〉……¥＊・＊" という文字列を作成してこのパス名に一致するファイルがすべて検索させる

List 8.3 SCHDIR.ASM

```
= 0004                          MAXLVL  equ     4 ……ディレクトリの最大の深さを4に設定
= 002B                          DTASIZE equ     43 …DTAに返されるディレクトリ情報用のバッファの大きさ
= 0002                          CHRLEN  equ     2  …MS-BASCOMは文字型変数の長さは2バイトで表
                                                   される. MS-BASICで用いる場合はここを1とする.
                                ;
                                ;   definition of stack frame
                                ;
0000                            stack   segment word public 'stack'
                                spframe struc
0000  ????                      bp_save dw      ?       ;bp register save
0002  ????????                  retadr  dd      ?       ;return address
0006  ????                      arg1    dw      ?       ;first argument
0008  ????                      arg2    dw      ?       ;second argument
000A  ????                      arg3    dw      ?       ;third argument
000C  ????                      arg4    dw      ?       ;fourth argument
000E                            spframe ends
0000                            stack   ends
                                ;
                                ;   definition of data area
                                ;
0000                            data    segment word public 'data'
                                ;
0000    D7 [                    dta     db      (MAXLVL+1)*DTASIZE dup (?)
            ??
               ]
00D7  ??                        start   db      ?
                                ;
00D8                            data    ends
                                ;
                                ;   program area
                                ;
0000                            code    segment byte public 'code'
                                        assume  cs:code,ds:data,ss:stack
                                        public  currentdir
                                        public  searchfirst,searchnext
                                ;                       BASICから参照できるように名前を宣言する
                                ;   current directory get
                                ;
0000                            currentdir      proc    far
0000  55                                push    bp
0001  8B EC                             mov     bp,sp
0003  B4 47                             mov     ah,47h
0005  B2 00                             mov     dl,0
0007  8B 76 06                          mov     si,arg1[bp]
000A  8B 74 02                          mov     si,CHRLEN[si]
000D  CD 21                             int     21h
000F  5D                                pop     bp
0010  CA 0002                           ret     2
0013                            currentdir      endp
```

BASICから渡されるパラメータのポインタで構成されるスタックの構造を定義する. パラメータの順序がCALL文と逆に定義されているのに注意

カレント・ドライブのカレント・ディレクトリ名をCALL文のパラメータの文字型変数にセットする. 文字型変数は64バイト以上の長さが必要

```
                                ;
                                ;   search first entry
                                ;
0013                            searchfirst     proc    far
0013  55                                push    bp
0014  8B EC                              mov     bp,sp
0016  B1 00                              mov     cl,0 ……最初のディレクトリ・エントリを検索するフラグをセットする
0018  E8 002B R                          call    sch_dir
001B  5D                                pop     bp
001C  CA 0008                           ret     8
001F                            searchfirst     endp
                                ;
                                ;   search next entry
                                ;
001F                            searchnext      proc    far
001F  55                                push    bp
0020  8B EC                              mov     bp,sp
0022  B1 01                              mov     cl,1 ……次のディレクトリ・エントリを検索するフラグをセットする
0024  E8 002B R                          call    sch_dir
0027  5D                                pop     bp
0028  CA 0006                           ret     6
002B                            searchnext      endp
                                ;
002B                            sch_dir proc    near
002B  1E                                push    ds
002C  B8 ---- R                          mov     ax,seg data  } DSレジスタのセット
002F  8E D8                              mov     ds,ax        }
0031  88 0E 00D7 R                       mov     start,cl ……最初のディレクトリ・エントリの検索かどうかのフラグをstartに格納

0035  8B 5E 08                           mov     bx,arg2[bp]  }
0038  26: 8B 07                          mov     ax,es:[bx]   } arg2で与えられた整数型変数(2バイト)がMAXLVL以内なら"levelok"へジャンプ
003B  3D 0004                            cmp     ax,MAXLVL    } そうでなければ－1のステータスを返す
003E  76 06                              jbe     levelok      }
0040  BA FFFF                            mov     dx,-1        }
0043  E9 00C7 R                          jmp     return       }

0046  BA 0000 R                 levelok:mov     dx,offset dta  }
0049  B4 2B                              mov     ah,DTASIZE     } 使用するDTAのセット
004B  F6 E4                              mul     ah             } 以前のディレクトリ情報を保存する必要があるのでarg2（BASIC側ではLEVEL）により別のDTAを使用する
004D  03 D0                              add     dx,ax          }
004F  8B FA                              mov     di,dx          }
0051  B4 1A                              mov     ah,1ah         }
0053  CD 21                              int     21h            }

0055  8B 5E 06                           mov     bx,arg1[bp]              } ファイルが見つからない場合には空文字が返されるように最初のバイトに00Hを入れる
0058  26: 8B 5F 02                       mov     bx,es:CHRLEN[bx]         }
005C  26: C6 07 00                       mov     es:byte ptr [bx],0       }

0060  33 D2                              xor     dx,dx ……ステータスのクリア
0062  80 3E 00D7 R 00                    cmp     start,0  } 次のディレクトリの検索ならば"tonext"へジャンプ
0067  75 29                              jne     tonext   }

0069  1E                                 push    ds
006A  8C C0                              mov     ax,es
006C  8E D8                              mov     ds,ax
006E  B4 4E                              mov     ah,4eh
0070  8B 76 0C                           mov     si,arg4[bp]
0073  26: 8B 54 02                       mov     dx,es:CHRLEN[si]
0077  B9 0017                            mov     cx,17h
007A  CD 21                              int     21h
007C  1F                                 pop     ds
007D  BA 0000                            mov     dx,0 ……ステータスのクリア
0080  72 41                              jc      no_more ……一致するファイルがなければno_moreへジャンプ

0082  BB 001E                   name_ck:mov     bx,1eh ……DTA＋1EHからファイル名がASCIZ文字列でセットされる
```

最初のディレクトリ・エントリの検索
arg4の文字列変数で渡されたパス名に一致するディレクトリ・エントリを検索する
システム属性のファイルや隠されたファイルも検索する

```
0085  8A 01                    name_lp:mov     al,[bx][di]
0087  3C 00                            cmp     al,0
0089  74 28                            je      nxtfile
008B  3C 2E                            cmp     al,'.'
008D  75 03                            jne     tonext
008F  43                               inc     bx
0090  EB F3                            jmp     name_lp

0092  80 0E 00D7 R 80         tonext: or        start,80h

0097  F6 45 15 10                      test    byte ptr [15h][di],10h
009B  74 04                            je      not_dir
009D  81 CA 0002                       or      dx,2

00A1  8D 75 1E                not_dir:lea     si,[1eh][di]
00A4  57                               push    di
00A5  8B 7E 06                         mov     di,arg1[bp]
00A8  26: 8B 7D 02                     mov     di,es:CHRLEN[di]
00AC  B9 000D                          mov     cx,13
00AF  FC                               cld
00B0  F3/ A4                     rep   movsb
00B2  5F                               pop     di

00B3  B4 4F                   nxtfile:mov     ah,4fh
00B5  CD 21                            int     21h
00B7  72 0A                            jc      no_more
00B9  F6 06 00D7 R 80                  test    start,80h
00BE  74 C2                            je      name_ck
00C0  EB 05 90                         jmp     return

00C3  81 CA 0001              no_more:or        dx,1

00C7  8B 5E 0A                return: mov     bx,arg3[bp]
00CA  26: 89 17                        mov     es:[bx],dx
00CD  1F                               pop     ds
00CE  C3                               ret
00CF                          sch_dir endp
                              ;
00CF                          code    ends
                              ;
                                       end
```

次にBASICのソースファイル〝TINYTREE．BAS〟をMS-BASCOMでコンパイルして，〝TINYTREE．OBJ〟を作成し，機械語サブルーチン〝SCHDIR．ASM〟をアセンブルして〝SCHDIR．OBJ〟を作成します．この2つのオブジェクトファイルをリンクすれば，実行ファイル〝TREE．EXE〟が作成されます．

MS-BASCOMでコンパイルして作成されるオブジェクトファイルは，通常ランタイムルーチンを内部割込みで呼び出す形式(内部割込み3FHでランタイムルーチンをコールするコードを生成する)になっていますが，ここではプログラムが独立して動くようにするために〝／O〟スイッチを指定(CALLでコードを生成し〝BASRUN．LIB〟とリンクする)しています(Fig.8．3)．

```
A>dir tinytree↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

TINYTREE BAS      3018  84-03-18   15:26……BASICのソースファイル(拡張子は".BAS")
        1 個のファイルがあります              コンパイルするためにはアスキーセーブされている必要がある
   407552 バイトが使用可能です

A>bascom↵……ベーシック・コンパイラを起動する

Microsoft BASIC Compiler
Version 5.35
(C)Copyright Microsoft Corp 1982 ┌──ソースファイルの指定."/O" スイッチを付けてランタイムルーチン
                                 │  (BASRUN.EXE)なしで実行可能とする
Source filename [.BAS]: tinytree/o↵
Object filename [TINYTREE.OBJ]:↵……オブジェクトファイル名はデフォルトを使用する
Source listing  [NUL.LST]: tinytree↵……リスティングファイルの指定

23791 Bytes Available
22106 Bytes Free

    0 Warning Error(s)
    0 Severe  Error(s)

A>dir tinytree↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

TINYTREE BAS      3018  84-03-18   15:26
TINYTREE OBJ      4111  84-03-18   15:28……オブジェクトファイル
TINYTREE LST      5273  84-03-18   15:28……リスティングファイル
        3 個のファイルがあります
   396288 バイトが使用可能です

A>dir schdir↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

SCHDIR    ASM      2215  84-03-18   15:27……アセンブラのソースファイル(拡張子は".ASM")
        1 個のファイルがあります
   396288 バイトが使用可能です

A>masm↵……アセンブラを起動する
The Microsoft MACRO Assembler , Version 1.25
 Copyright (C) Microsoft Corp 1981,82,83

Source filename [.ASM]: schdir↵……BASICとリンクさせる"SCHDIR"をソースファイルに指定
Object filename [schdir.OBJ]:↵……オブジェクトファイル名はデフォルトを使用する
Source listing  [NUL.LST]: schdir↵……リスティングファイルの指定
Cross reference [NUL.CRF]: ↵

Warning Severe
Errors  Errors
0       0

A>dir schdir↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

SCHDIR    ASM      2215  84-03-18   15:27
SCHDIR    OBJ       408  84-03-18   15:29……オブジェクトファイル
SCHDIR    LST      7144  84-03-18   15:29……リスティングファイル
        3 個のファイルがあります
   388096 バイトが使用可能です
```

```
A>link↵ ……リンカを起動する

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: tinytree schdir …前に作成した"TINYTREE.OBJ"と"SCHDIR.OBJ"をリンクする
Run File [TINYTREE.EXE]: tree↵ ……実行型ファイルの指定
List File [NUL.MAP]: tree↵ ……マップファイルの指定
Libraries [.LIB]:↵ ……ライブラリは指定しなければ"BASRUN.LIB"が用いられる

A>dir tree↵

 ドライブ A: のボリュームラベルは HANDBOOK
 ディレクトリは A:¥

TREE      MAP       904   84-03-18   15:30 ……マップファイル
TREE      EXE     14888   84-03-18   15:30 ……実行型ファイル
        2 個のファイルがあります
   371712 バイトが使用可能です

A>■
```

Fig. 8.3　コンパイルおよびリンク手順

この"TREE.EXE"を実行した結果をFig. 8.4に示します.

```
A>tree↵ ……"TREE.EXE"の実行

Root?↵ ……リターンのみを入力するとカレント・ディレクトリ(現在はルート・ディレクトリにいる)から表示される

¥ -------------- IO.SYS          } 隠されたファイルも表示される
                ¦- MSDOS.SYS    }
                ¦- COMMAND.COM
                ¦- BIN ------------ DEBUG.COM
                ¦                   ¦- EDLIN.COM
                ¦                   ¦- EXE2BIN.EXE
                ¦                   ¦- LINK.EXE
                ¦                   ¦- MASM.EXE
                ¦                   ¦- DUMP.COM
                ¦                   ¦- SUBDIR --------- ……このディレクトリには". ."と". ."
                ¦                                         ファイルしかないので何も表示されない
                ¦- COM ------------ MORE.COM
                ¦                   ¦- KEY.COM
                ¦                   ¦- PRINT.COM
                ¦
                ¦- AUTOEXEC.BAT
                ¦- CONFIG.SYS
                ¦- SUBDIR1 -------- SUBDIR2 -------- TEST
                ¦                   ¦
                ¦                   ¦- FACT.C
                ¦                   ¦- LANGUAGE.TXT
                ¦                   ¦- TYPEFILE.BAT
                ¦
                ¦- CAL ------------ FACT.C
                ¦                   ¦- FIBBO.C
                ¦                   ¦- ACKERF.C
                ¦                   ¦- ACKERF.MAP
                ¦                   ¦- ACKERF.OBJ
                ¦                   ¦- ACKERF.EXE
                ¦
                ¦- TXT ------------
                ¦
                ¦- ASMWORK -------- FUNC-00.ASM
                                    ¦- FUNC-06.ASM
                                    ¦- FUNC-0B.ASM
```

```
A>tree ↵ ……再び "TREE.EXE" を起動

Root? bin ↵ ……上で表示したディレクトリのBINのみを表示させてみる

BIN ------------ DEBUG.COM
                 |- EDLIN.COM
                 |- EXE2BIN.EXE
                 |- LINK.EXE
                 |- MASM.EXE
                 |- DUMP.COM
                 |- SUBDIR ---------


A>■
```

Fig. 8 . 4　TREE . EXE 実行結果

コンパイルの手順の中では〝／O〞スイッチを使用していますが，このほかにもコンパイル時に各種スイッチを付けることができます.たとえば〝／D〞スイッチを指定すると，TRON, TROFFを使用して，実行されている行番号を表示させることができるので，デバッグによく用いられます.

8.2.4　リンク時の注意事項

MS-BASCOMでコンパイルしたオブジェクトモジュールと機械語のモジュールをリンクする場合には，以下の点に注意する必要があります.

(1) BP, SP, DS, SSレジスタは破壊してはならない
(2)機械語のプログラムは〝PROC FAR〞ではじまらなければならない

(1)については説明の必要はないと思いますが，(2)については少し説明を加えておきます.

CALL，CALLS文で機械語のプログラムを呼び出す場合，Fig.8 . 5に示すようにCALL(CALLS)文のパラメータは，そのポインタがスタックに積まれた後に，FAR CALL命令で機械語のサブルーチンを呼び出すため，機械語サブルーチンは〝PROC FAR〞ではじまらなければならないのです．また，サブルーチンから戻るときには，スタックに積まれたパラメータのポインタを取り出す必要があるため，CALL命令では〝RET(パラメータの数)×2〞，CALLS命令では〝RET(パラメータの数)×4〞を行ってBASICのオブジェクトモジュールへ制御を戻さなければなりません.

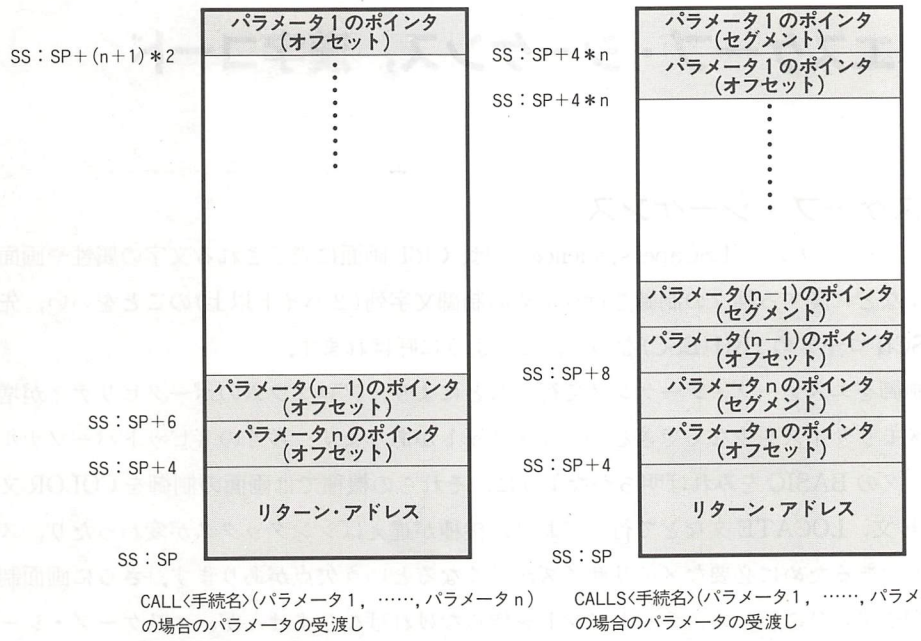


Fig. 8．5　CALL，CALLSのスタックにおけるパラメータ

付録　エスケープ・シーケンス，漢字コード

1.　エスケープ・シーケンス

エスケープ・シーケンス(ESCape sequence)とは，CRT 画面に表示される文字の属性や画面のスクロールなどに関する種々の制御を行うための制御文字列(2バイト以上)のことをいい，先頭文字が ASCII コードの 1BH(ESC)なので，このように呼ばれます．

画面の制御をエスケープ・シーケンスで行うことにより，プログラムのポータビリティが増すとか所要メモリサイズを少なくできるというメリットがあります．多くの8ビットパーソナル・コンピュータの BASIC をみれば明らかなように，それらの機種では画面の制御を COLOR 文や CONSOLE 文，LOCATE 文などで行っており，機種が違えばシンタックスが変わったり，ステートメントであるために必要なメモリサイズが多くなるという欠点があります．さらに画面制御の機能を増やすたびに新しいステートメントを作らなければなりませんが，エスケープ・シーケンスを利用することによりこれらの欠点をかなり克服することができます．

エスケープ・シーケンスでどんな制御や機能を実現するかについては ANSI(米国規格協会)および ISO(国際標準化機構)で標準的に推奨されるべきものが制定されており，MS-DOS でもこの規格を基に MS-DOS 標準エスケープ・シーケンスとでも呼べるものが決められています．しかし ANSI や ISO の規格はかなり多機能で複雑であるため，MS-DOS では，あまりハードウェアに依存せずよく使われるものを選んで Table 1 に示す内容を標準エスケープ・シーケンスとしてあります．ただし，これはあくまでも標準推奨のものであり，メーカーによっては機能を追加したり，ハードウェアの制約によって機能の削除が行われています．

なお，Table 1 で "ESC" は ESC コードを表し，pn，pl，pc，ps は 10 進数で表した数字列のことを指します．省略したときは各エスケープ・シーケンスにより適宜解釈されます．

Table 1　MS-DOS標準 エスケープ・シーケンス一覧

エスケープ・シーケンス	機　能
ESC［pl;pcH	カーソルをpl行pc列へ移動する
ESC［pl;pcf	カーソルをpl行pc列へ移動する(ESC Hと同じ)
ESC［pnA	カーソルを同一桁でpn行上へ移動する
ESC［pnB	カーソルを同一桁でpn行下へ移動する
ESC［pnC	カーソルを同一行でpn桁右へ移動する
ESC［pnD	カーソルを同一行でpn桁左へ移動する
ESC［pnM	カーソルの位置する行からpn行を削除する

エスケープ・シーケンス	機　能
ESC［6n	カーソルの位置を直後のコンソール入力にて知らせる．その形式は"ESC［pl;pcR"
ESC［s	カーソル位置とその位置の文字の属性をセーブする
ESC［u	セーブしたカーソル位置と属性をもとへ戻す
ESC［2J	スクリーンを消去してカーソルをホーム位置へ移動する
ESC［K	カーソル位置から行末までを消去する
ESC［ps;...;ps m	表示文字の属性を指定する(psは複数個指定できる) ps　機能 0　すべての属性をデフォルトに戻す 1　強調(HIGH LIGHT or BOLD) 4　下線 5　ブリンク 7　リバース 8　シークレット 30　黒 31　赤 32　緑 33　黄色 34　青 35　マゼンタ 36　水色 37　白 40　背景色　黒 41　背景色　赤 42　背景色　緑 43　背景色　黄色 44　背景色　青 45　背景色　マゼンタ 46　背景色　水色 47　背景色　白

2.　漢字コード

MS-DOS は米国で開発された DOS ですが，日本でそのユーザーを増やすためには何らかの方法で日本語を扱えなければなりません．そこで決められたのが〝MS-DOS 標準漢字仕様〟です．しかしこの仕様では MS-DOS で使用する漢字コードの体系が決められているにすぎず，漢字の入力については，MS-DOS をインプリメントしているメーカーにより大きく異なっているのが現状です．

漢字コードの規格はすでに JIS で〝JIS 漢字コード〟が制定されています．しかしこのコード体系をそのまま ASCII コードや 8 ビットの JIS コードの体系に持ち込んだ場合，漢字コード(2 バイト)と普通の 8 ビットを区別するために KSI(Kanji Shift In)，KSO(Kanji Shift Out)コードが必要になります．つまり，漢字コード列を KSI と KSO で挟むわけですが，このとき 2 つのシフトコードの扱いには気を付けなければなりません．たとえば，2 つ以上の漢字コード列を連結するときは接続部分の KSI と KSOを除去しないと文字列がどんどん長くなってしまいますし，2 つ以上に分割するときは KSI と KSO を付加してやらなければなりません．また，KSI と KSO はネストしないようにしなければなりませんし，ある文字列中の指定された位置にある 1 バイト

が漢字コードの一部か，それとも単なる1バイト系の文字であるかどうかの判定もかなり手間がかかります．

これらの欠点を克服するために，MS-DOSは漢字コード体系としてJISコードを少々アレンジした〝シフトJISコード〟を採用しています(Fig. 1)．このコード体系では，漢字コードの第1バイトが80H～9FHまたはE0H～FFHの範囲内にはいっているので，漢字コードかどうかの判定が簡単に行え，かつKSIやKSOのようなむだな情報がいりません．ちなみに，シフトJISコードの第1バイトの部分は，JISコードでは未定義領域となっています(この部分は従来の8ビットマシンではグラフィックパターンがはいっていた)．なお，機種によってはシフトJISコードのほかに，JISコードや区点コード，内部連続コードと呼ばれるものをサポートしているものもありますが，どのMS-DOS上でも正しく動作するプログラムを作るには，シフトJISコードのみを考慮しておけばよいでしょう．

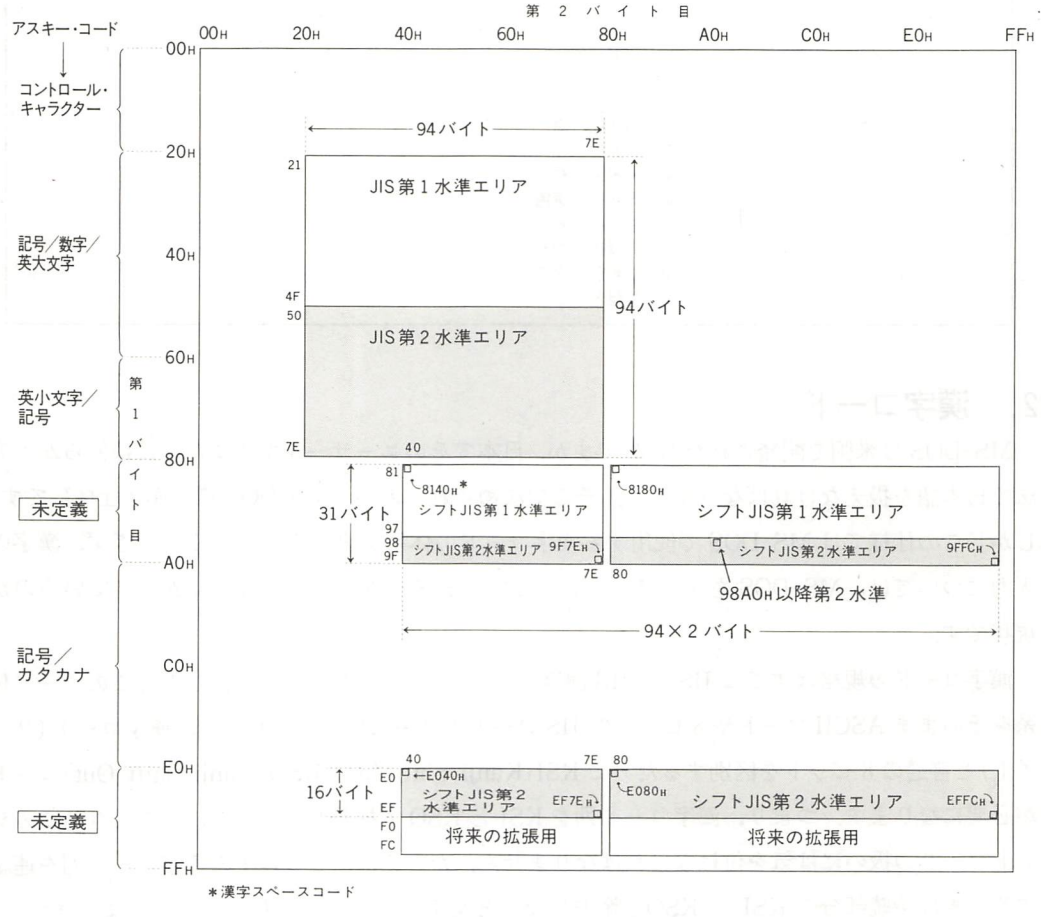


Fig. 1　JISコードとシフトJISコード

索　引

A

ASCIZ文字列……246
AT……368
AUX……25,30,58,102

B

BASIC……496
BATファイル……26,69,218
BLOCK／CHARACTER……314
BPB……320
BUFFERS……305
BUILD BPB……332

C

C言語……491
CLOCK……25,308,315
CTRL-Cアドレス……234
COMファイル……26,84,218
COMMAND．COM……21,112,215,218
COMSPEC……305
CON……25,30,58,59,103
CONFIG．SYS……214,304
COUNTRY……306
CP／M……15,262,346,373
CREF……76,79

D

DEBUG……77,180
DPB……324
DTA……276,396,397,417,449

E

EDLIN……33,76,79
EXEファイル……218
EXE2BIN……76,84

F

FAT ID……347
FCB……259,275
FILES……304
FLUSH……335

I

I／Oのリダイレクト……223,274
I／Oリクエスト……316,326
INIT……329
INT……232
IO-CONTROL……314
IO．SYS……209
IO．SYSルーチン……212

L

LIB……76,82
LINK……76

M

MASM……76
MEDIA CHECK……331
MEMORY……368
MSDOS．SYS……20,208

N

NO WAIT……334
NON DESTRUCTIVE INPUT……334
NON-IBM FORMAT……314
NUL……25,59,315

O

OFFSET……370

P

PRN……25,58
PSP……241
PTR……369
PUBLIC……367

R

RS-232C……30,58,102,307

S

SEG……370
SHELL……305
SHORT……370
SIZE……371
SPECIAL……315
STACK……367
STATUS……335
STI……314
STO……314
SYSINITルーチン……212

T

TPA……215

U

UNIX……15

X

XENIX……15

ア

エラーコード……257,375
オブジェクトファイル……79

カ

階層的ディレクトリ……51,336
拡張FCB……267,399
拡張子……25
カレントブロック……264
カレントレコード……264
カレント・ドライブ……31,396
環境……73,135,221,246,476
外部参照……81,386
キャラクタ型デバイス……308
擬似命令……351
クラスタ……342,343
クラスタ番号……340
クロスリファレンスファイル……77,79
グループ……237
コマンドパケット……316,326
コンソール……273,290
コントロール・キャラクタ……28

サ

再配置因子……285
システムコール……271
終了アドレス……234
ストラクチャ……350,354
ストラテジ・エントリポイント……315,317
セクタ……342
セグメント……237,352,363
相対レコード・フィールド……259,422

タ

ダーティバッファ……292
テンポラリファイル……43,228
ディレクトリ……336,467
ディレクトリ・エントリ……336,396,485
デバイス……91,275
デバイス情報……280
デバイス・ドライバ……281,307
デバッグ……85

ナ

名前……349
内部割込み……231
入力バッファ……274

ハ

パイプ……62,225
パス……52,123,220
パス名……52,91
バックアップ……29
パックネーム……482
バッチファイル……72
バッファリング……290
パラグラフサイズ……469
パラメータ……71,90,157,159,181
標準エラー出力……224,269
標準入出力……224,269
標準プリンタ……224,269,379
標準補助装置……224,269,379
ファイル……30,336,404,407,409,440,441,442
ファイル管理……50
ファイルサイズ……340,419
ファイル属性……283,339,454
ファイル名……24,91,338,425
ファイル・ハンドル……268,279,298,461,464
フィルタ……61,115,121,137
ブート……211,212
プライベート……367
プログラムセグメントプリフィックス……241
ブロック型デバイス……308
プロシージャ……237,350
プロセス……249,301
プロセス管理……284

マ

マクロ機能……357
メディア・ディスクリプタバイト……322,324
メモリ管理……287,303
メモリ・コントロールブロック……469,470
メモリマップ……211
メモリモデル……237
モジュール……209

ラ

ラベル……349,356
ライブラリファイル……77,81

リアルタイムクロック……307
リクエストヘッダ……326
リダイレクト……59,224
リンク……82,309,504
レコード……350,355
レコードサイズ……259,264

ワ

ワイルドカード……26,260
割込みベクトルテーブル……211
割込みエントリ・ポイント……315,317

＃……25
．……54,340
．＄＄＄……26,43
．．……54,340
．COM……25
．EXE……25
．SYS……25
＜……60,226
＞……60,226
？……26,159
＊……26

００Ｈ　プログラムの終了……376
０１Ｈ　キーボード入力とコンソールのエコ-376
０２Ｈ　文字の出力……377
０３Ｈ　補助入力……378
０４Ｈ　補助出力……379
０５Ｈ　プリンタ出力……379
０６Ｈ　コンソールとの直接入出力……381
０７Ｈ　コンソールからの直接入力……383
０８Ｈ　エコーなしのキーボード入力……383
０９Ｈ　文字列の出力……384
０ＡＨ　バッファード・キーボード入力……388
０ＢＨ　キーボードのステータス・チェック…390
０ＣＨ　バッファを空にしてキーボード入力‥394
０ＤＨ　ディスクのリセット……395
０ＥＨ　ドライブの選択……396
０ＦＨ　ファイルのオープン……401
１０Ｈ　ファイルのクローズ……402
１１Ｈ　最初のディレクトリ・エントリの検索…396
１２Ｈ　次のディレクトリ・エントリの検索……397
１３Ｈ　ファイルの削除……407
１４Ｈ　シーケンシャルな読出し……403
１５Ｈ　シーケンシャルな書込み……403
１６Ｈ　ファイルの作成……404
１７Ｈ　ファイル名の変更……409
１９Ｈ　カレント・ドライブ番号の読出し……412
１ＡＨ　ディスク転送アドレスの設定……417
２１Ｈ　ランダムな読出し……418
２２Ｈ　ランダムな書込み……419
２３Ｈ　ファイルサイズの計算……419
２４Ｈ　相対レコード・フィールドの設定……422
２５Ｈ　割込みベクトルの設定……429
２７Ｈ　ランダムなブロック読出し……423
２８Ｈ　ランダムなブロック書込み……424
２９Ｈ　ファイル名の解析……425
２ＡＨ　日付の読出し……391
２ＢＨ　日付の設定……391
２ＣＨ　時刻の読出し……392
２ＤＨ　時刻の設定……392
２ＥＨ　ベリファイフラグの設定……431
２ＦＨ　ディスク転送アドレスの読出し……449
３０Ｈ　DOSのバージョン番号の読出し…412
３１Ｈ　プログラムの常駐終了……474
３３Ｈ　ブレーチェックの制御……434
３５Ｈ　割込みベクトルの読出し……430
３６Ｈ　ディスクの残り容量の読出し……412
３８Ｈ　国別情報の読出し……413
３９Ｈ　サブ・ディレクトリの作成……436
３ＡＨ　サブ・ディレクトリの削除……436
３ＢＨ　カレント・ディレクトリの変更……437
３ＣＨ　ファイルの作成……440
３ＤＨ　ファイルのオープン……441
３ＥＨ　ファイルのクローズ……442
３ＦＨ　ファイル／デバイスの読出し……442
４０Ｈ　ファイル／デバイスの書込み……443
４１Ｈ　ファイルの削除……447
４２Ｈ　ファイル・ポインタの移動……450
４３Ｈ　ファイルの属性の変更……454
４４Ｈ　デバイスに対するI／Oコントロール‥456
４５Ｈ　ファイル・ハンドルのコピー……461
４６Ｈ　指定したファイル・ハンドルへのコピー 464
４７Ｈ　カレント・ディレクトリの読出し……467
４８Ｈ　メモリブロックの割当て……469
４９Ｈ　メモリブロックの解放……470
４ＡＨ　メモリブロックのサイズの変更……471
４ＢＨ　プログラムのロードと実行……474
４ＣＨ　プロセスの終了……476
４ＤＨ　子プロセスの終了コードの読出し…477
４ＥＨ　一致するファイルの検索……481
４ＦＨ　次に一致するファイルの検索……482
５４Ｈ　ベリファイフラグの読出し……432
５６Ｈ　ディレクトリ・エントリの移動……485
５７Ｈ　ファイルの時間情報の読出し／設定‥488

参 考 文 献

1） 各社 MS-DOS マニュアル
2）「応用 CP／M」　村瀬康治著　アスキー発行
3）「標準 UNIX ハンドブック」　Richard Gauthier 著　アスキー発行
4）「THE UNIX PROGRAMMING ENVIROMENT」　Brian W. Kernighan, Rob Pike 著
PRENTICE-HALL 発行
5）「MS-DOS読本」　菅木真治著　アスキー発行
6）「ザ・8086ブック」　レクター著　産報出版発行
7）「16bit Personal Computer」　打越浩幸他著　アスキー発行
8）「iAPX86ファミリ・ユーザーズマニュアル」　インテルジャパン発行
9）「iAPX86マクロアセンブリ言語プログラミングマニュアル」　インテルジャパン発行
10）「入門 MS-DOS」　村瀬康治著　アスキー発行

著 作 協 力

打越浩幸　榊　正憲
川名章史　嶋谷洋一
小松克行

標準 MS-DOS ハンドブック

1984年7月10日　初版発行
1989年3月11日　第1版第18刷発行
定価3,500円

編　著　アスキー出版局
発行者　塚本慶一郎
発行所　株式会社アスキー
〒107 東京都港区南青山6-11-1 スリーエフ南青山ビル
振　替　東京4-161144
TEL（03)486-7111(大代表)
情報TEL（03)498-0299(ダイヤルイン)
出版営業部TEL（03)486-1977(ダイヤルイン)

本書は著作権法上の保護を受けています．本書の一部あるいは全部について（ソフトウェア及びプログラムを含む），株式会社アスキーから文書による許諾を得ずに，いかなる方法においても無断で複写，複製することは禁じられています．

編集担当　土屋信明
表紙担当　郷　啓子
印刷　壮光舎印刷株式会社

ISBN4-87148-742-3 C3055 ¥3500E

定価3,500円

ISBN4-87148-742-3 C3055 ¥3500E

MS-DOS
HANDBOOK

標準MS-DOSハンドブック

アスキー出版局編著

ASCII
ハンドブックス